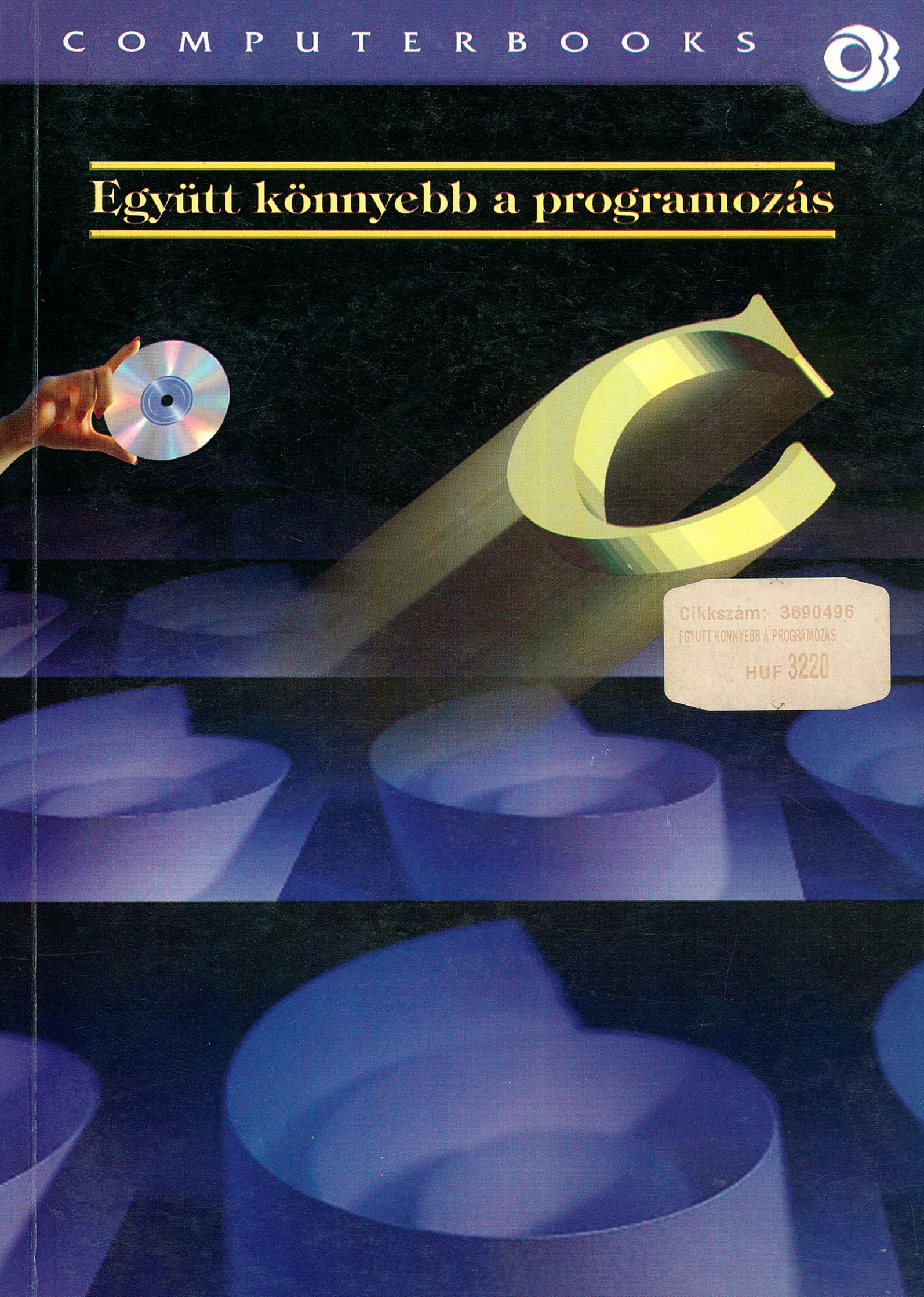
COMPUTERBOOKS

Együtt könnyebb a programozás



Cikkszám: 3690496
EGYÜTT KÖNNYEBB A PROGRAMOZÁS
HUF 3220

MINDENT EGY HELYEN

authoring, ÚJ

mastering, sokszorosítás...

DVD5, DVD9, DVD10

VTCD VIDEOTON Kompaktlemez-gyártó Kft.

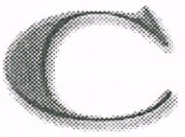
...továbbra is az Ön partnere vagyunk CD, MC gyártásban.

H-8000 Székesfehérvár, Pf.:175.
Tel.:(22) 533-571, Fax:(22) 533-077
vtcd@vtcd.hu www.vtcd.hu

DVD Authoring studio:
H - 1021 Budapest, Hűvösvölgyi út 54,
Tel. : (1) 392-1217, Fax : (1) 392-1238
E-mail: authoring@vtcd.hu www.vtcd.hu

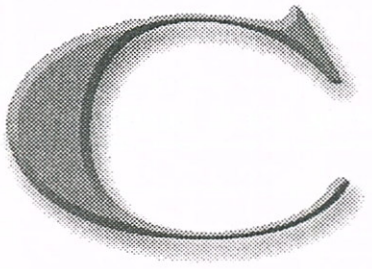
TÜV Rheinland EUROQUA EN ISO 9001:2000 ZN 75.100.8943

Együtt könnyebb a programozás



Benkő Tiborné
Dr. Poppe András

Együtt könnyebb a programozás



LEKTOR
TÓTH BERTALAN

COMPUTERBOOKS
BUDAPEST, 2004

A könyv készítése során a Kiadó és a Szerzők a legnagyobb gondossággal jártak el. Ennek ellenére hibák előfordulása nem kizárható.
Az ismeretanyag felhasználásának következményeiért sem a Szerzők, sem a Kiadó felelősséget nem vállal.

Minden jog fenntartva. Jelen könyvet vagy annak részleteit a Kiadó engedélye nélkül bármilyen formátumban vagy eszközzel reprodukálni, tárolni és közölni tilos.

©Benkő Tiborné, Dr. Poppe András, 2004. október
Társszerző: Tóth Bertalan

ISBN: 963 618 325 2

©Kiadó: ComputerBooks Kiadó Kft.
1126 Bp., Tartsay Vilmos u. 12.
Telefon: 375-1564; Tel/Fax: 375-3591
E-mail: info@computerbooks.hu
http://www.computerbooks.hu

Felelős kiadó: ComputerBooks Kft. Ügyvezetője
Borítóterv: Székely Edith

Nyomdai munkák: Pressman Bt.

Tartalomjegyzék

Bevezetés 1

1. A C program szerkezete 3
1.1 A legegyszerűbb C program 3
1.2 Írás képernyőre 3
1.3 Több modulból álló C program 5

2. Alaptípusok, változók és konstansok 7
2.1 A C nyelv típusai 7
2.2 Szabványos input és output 7
2.2.1 A printf () függvény 7
2.2.2 A scanf() függvény 10
2.2.3 A getchar() és a putchar() függvények 12
2.2.4 A gets() és a puts() függvények 12
2.3 A C nyelv típusainak bemutatása 14
2.3.1 Típuselőírások, típusmódosítók 14
2.3.2 A felsorolt típus (enum) 14
2.3.3 Saját típusok előállítása 15
2.3.4 Konstansok a C nyelvben 16
2.4 Értékek, címek és mutatók 16
2.4.1 Balérték és jobbérték 16
2.4.2 Ismerkedés a mutatókkal 17
2.4.3 Mutatók és a dinamikus memóriahasználat 18
2.4.4 A void* típusú általános mutató 19
2.4.5 Többszörös indirektségű mutatók 20
2.5 Matematikai függvények 20

3. Operátorok és kifejezések 23
3.1 Precedencia és asszociativitás 23
3.1.1 A precedencia szabály 24
3.2 Mellékhatások és a rövidzár 26
3.3 Értékadó operátorok 27
3.4 Léptető (inkrementáló/dekrementáló) operátorok 27
3.5 Bitműveletek 29
3.5.1 Biteltoló műveletek 30
3.6 A sizeof operátor 31
3.7 A vessző operátor 32

3.8 A fetételes operátor 32
3.9 Statikus és dinamikus helyfoglalású változók használata 33
3.9.1 Mutatók használata 34
3.10 Konstansok használata 35
3.11 Saját típus előállítása 36

4. A C nyelv utasításai 37
4.1 Utasítások és blokkok 37
4.2 Az if utasítás 37
4.3 A switch utasítás 41
4.4 Ciklusutasítások 45
4.4.1 A while ciklus 45
4.4.2 A for ciklus 48
4.4.3 A do-while ciklus 50

5. Származtatott típusok 53
5.1 Tömbök, sztringek és mutatók 53
5.1.1 Egydimenziós tömbök 53
5.1.2 Többdimenziós tömbök 56
5.1.3 Karaktersorozatok (sztringek) 58
5.2 Felhasználó által definiált adattípusok 60
5.2.1 A struct struktúratípus 61
5.2.2 Struktúratömbök 62

6. Függvények 67
6.1. A függvények definiálása 67
6.2 Különböző típusú paraméterek 68
6.2.1 Aritmetikai típusú paraméterek 68
6.2.2 Felhasználói típusú paraméterek 71
6.2.3 Tömbök átadása függvénynek 75
6.2.4 Sztringparaméterek 79
6.2.5 Változó hosszúságú paraméterlista 81
6.2.6 A main() függvény paraméterei és visszatérési értéke 82
6.2.7 Rekurzív függvények használata 83

7. Modulok használata 85
7.1 Az extern tárolási osztály alkalmazása 85
7.2 Több modulból álló C program 86

8. Láncolt lista 89
8.1 Előre-hátra láncolt lista 89

9. Algoritmusok programozása 95
9.1 Osztók, legnagyobb közös osztó, legkisebb közös többszörös meghatározása 95

9.2 Keresési és rendezési algoritmusok 101
9.2.1 Keresési algoritmusok 101
9.2.2 Rendezési algoritmusok 103

10. Szövegfájlok kezelése 111
10.1 Szöveges állományba való írás és olvasás 113

11. Bináris állomány I/O műveletei 119
11.1 Bájtonkénti feldolgozás 120
11.2 Numerikus értékek bináris fájlba való tárolása 122
11.3 Struktúrák bináris állományba történő írása és olvasása 124

12. Numerikus módszerek 131
12.1 Nemlineáris egyenlet numerikus megoldása 131
12.2 Numerikus integrálás (numerikus kvadratúra) 134
12.3 Interpoláció 139
12.4 Lineáris egyenletrendszer megoldása 142
12.4.1 Lineáris egyenletrendszer megoldása LU-dekompozícióval 143
12.4.2 Mátrixinvertálás 152

F1. A Dev-C++ fejlesztői környezet használata 155
F1.1 A C program állományai 155
F1.2 Új forrásállomány és project létrehozása 155
F1.3 A program futtatása 159
F1.4 Project betöltése 160
F1.5 Hogyan működik a fejlesztői környezet 160

F2. C óravázlatok 163
1. Gyakorlat 165
2. Gyakorlat 172
3. Gyakorlat 180
4. Gyakorlat 185
5. Gyakorlat 192
6. Gyakorlat 198
7. Gyakorlat 215
8. Gyakorlat 227
9. Gyakorlat 237
10. Gyakorlat 250
11. Gyakorlat 253

F3. A CD-melléklet tartalomjegyzéke 269

Tárgymutató 271

Bevezetés

A *ComputerBooks Kiadó* új, "*Együtt könnyebb a programozás*" sorozatának második kötete ez a C tankönyv, ami jól segítheti a C nyelv alapjainak önálló elsajátítását. A művet azonban az oktatók is haszonnal forgathatják, hisz a tananyag az alapkönyvet kiegészítő, 11 gyakorlati foglalkozásra bontva, óravázlatként is felhasználható.

Joggal vetődik fel a kérdés, hogy miért éppen a C nyelvet választottuk e kötet témájaként. A válasz magától adódik, hisz a legtöbb napjainkban használt programozási nyelv (C++, Java, C#, PHP, Phyton, Perl stb.) alapjait a C nyelvben kell keresnünk. A C nyelv, az újabb nyelvekhez képest, egy egyszerű, jól tanítható, illetve tanulható programnyelv, melyről tetszőleges irányba tovább lehet lépni.

Könyvünk alapvető célja a C nyelvű programok készítésének feladatokon keresztüli megismertetése. A feladatok megoldását rövid elméleti magyarázat vezeti be, azonban részletesebb leírást talál az Olvasó a *ComputerBooks Kiadó "Programozzunk C nyelven!"* című kiadványában.

A könyv fejezeteiben a programkészítés módszerének elsajátításához mutatunk be gyakorló feladatokat, melynek megértése, illetve a kiírt feladatok megoldása nagyban hozzájárul az önálló tanuláshoz. Csak röviden világítunk rá azokra az elméleti ismeretekre, amelyek feltétlenül szükségesek a feladatok megoldásához.

Fontos része a műnek a **függelék**, és ez a megállapítása nemcsak az oldalszámra, hanem a tartalomra is vonatkozik. Az F2. függelék olyan oktatási anyagot tartalmaz, amely gyakorlati foglalkozásonként csoportosítja a különböző témaköröket. Az itt bemutatott feladatlapok bővített változata megtalálható a CD-mellékleten PDF-formátumban. A feladatlapok felölelik a könyv teljes tartalmát, azonban további feladatokon keresztül mutatják be a C lehetőségeit, rendszerezetten végighaladva az alapismereteken. Minden gyakorlati foglalkozás elején röviden ismertetjük az aktuális témakört és a hozzá tartozó mintafeladatokat.

A kiírt gyakorlati feladatok három részre oszthatók:

- Némely feladathoz ajánlott deklarációt és definíciót javasolunk, illetve megadjuk az elkészítendő függvények prototípusát. Egyes feladatoknál a várható futási eredményt is közöljük.
- Vannak olyan feladatok is, amelyek részmegoldást tartalmaznak, azonban a befejezéshez bizonyos programrészeket önállóan kell elkészíteni.
- Néhány feladatnak csak a kiírása szerepel – ezeket teljesen önállóan kell megoldani.

A könyvünk a C programozást tanítja, így a példaprogramokban, feladatmegoldásokban elsősorban az adott témakör bemutatását céloztuk. Az egyszerűség és a jól érthetőség érdekében – részben – lemondtunk a komoly fejlesztéseknél nélkülözhetetlen ellenőrzésekről és hibakezelési fogásokról.

A CD-melléklet a könyv anyagához tartozó programok forráskódján és néhány elektronikus formátumú fejezeten túlmenően egy szabadon terjeszthető C/C++ fejlesztőeszközt is tartalmaz. A ***\DevCpp\devcpp4970-gcc295.exe*** állományával ***Windows*** alatt futtatható, ***ANSI C/C++*** fejlesztői környezetet telepíthető a rendszerünkbe.

Minden tanulási folyamat fontos része az önellenőrzés, amihez a CD-melléklet ***\Teszt*** könyvtárában elhelyezett, a *Windows98/2000/XP* alatt működő tesztprogram, illetve az itt található feladatsorok nyújtanak hathatós segítséget.

Köszönetnyilvánítás

Ezúton szeretnénk köszönetet mondani a Budapesti Műszaki és Gazdaságtudományi Egyetem Természettudományi Kar két mérnök-fizikus hallgatójának, ***Udvarhelyi Anikónak*** és ***Wacha András Ferencnek***, akik a CD-mellékleten található oktatási anyagot áttanulmányozták, és értékes észrevételeket, javaslatokat tettek.

Továbbá, köszönjük ***Lehel György*** programozónak a függelékben kidolgozott C óravázlatokhoz fűzött építő megjegyzéseit.

A Szerzők

1. A C program szerkezete

A C program szerkezetét egyszerű példákon keresztül mutatjuk be.

1.1 A legegyszerűbb C program

Minden C program egy vagy több függvényből épül fel. A függvények általában meghatározott feladatot látnak el. A függvények közül egyet kötelezően a ***main*** névvel kell ellátni. A program végrehajtása ennek a függvénynek az aktiválásával kezdődik.

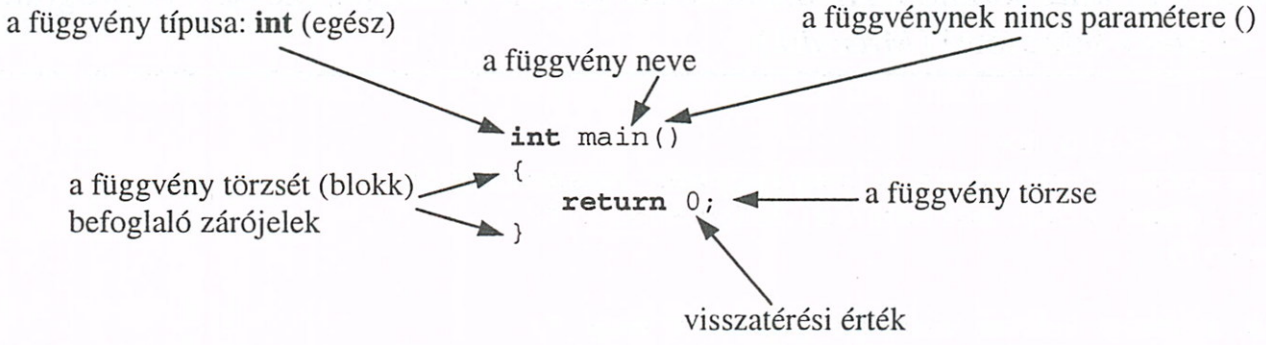
A legegyszerűbb C program egy üres ***main()*** függvény. A ***main()*** függvénynél kerek nyitó és záró zárójelet még akkor is ki kell tenni, nem rendelkezik paraméterrel. A függvény törzsét a nyitó és csukó kapcsos zárójel fogja közre (*Egyszeru*).

```
main()
{
}
```

A függvény jellemzője a visszatérési értéke, melynek típusát a függvényfejben, a függvény neve előtt kell megadni. Ha nem adunk meg típust, bizonyos implementációk figyelmeztetést (*warning*), más implementációk hibajelzést (*error*) adnak.

Visszatérési értéket a C függvény a **return** utasítás utáni kifejezés kiértékelésével határozza meg. A ***main()*** függvénynél a visszatérési értéket hibajelzésre használják, 0 visszatérési érték esetén hibátlan, 1 visszatérési érték esetén hibás volt a végrehajtás eredménye. *Megjegyzés*: más alkalmazások is hívhatnak C programot.

A legegyszerűbb C program egyetlen üres ***main()*** függvény:



1.2 Írás képernyőre

A C programban a képernyőre való íráshoz, így egy szöveg kiírásához is a ***printf()*** adatfolyamkezelő függvényt használjuk, melynek deklarációját (prototípusát) az *STDIO.H* állomány tartalmazza. A programba *#include* előfordító utasítással építjük be az *STDIO.H* (**St**andard **I**nput **O**utput) állományt < > jelek között megadva a nevét.

Ha elfelejtenénk a programba beépíteni, akkor néhány C fordítónál figyelmeztető jelzést kapunk, vagy hibásan fordul le a program.

A ***printf()*** függvény formátum szerint ír ki, a formátumot az ún. formátumsztring határozza meg. A formátumsztringet kettős idézőjelek "" között kell megadni. A benne megadott szöveg kiíródik, új sort a '\n' karakterrel emelhetünk.

Ahhoz, hogy a program futtatásakor az eredmény kiíratása után ne térjünk azonnal vissza a fejlesztői környezetbe, fel kell függesztenünk a program futását. Ebből a célból a programból való kilépés előtt meg kell hívnunk a ***getchar()*** függvényt, amely *<Enter>* lenyomásáig várakozik.

Írjunk programot, amely szöveget ír ki a képernyőre! A program várjon az *<Enter>* leütésére! (*Elso*)

```
#include <stdio.h>
int main()
{
  printf("Nem nehez C nyelven programozni!");
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
Nem nehez C nyelven programozni!
```

Írjunk programot, amely szöveget ír ki a képernyőre, és a szöveg közben, valamint a végénél is sort emel! Az első és utolsó sornál szóközzel pozicionál a második sor tabulátorral ('\t') kiírt sorához. Az utasításokat lássuk el megjegyzéssel! A program várjon az *<Enter>* leütésére! (*Masodik*)

```
#include <stdio.h>
/* A program bemutatja a megjegyzés használatát,
melyet lehet több sorban is folytatni. */
int main()
{
  /* a szöveg három sorban jelenik meg */
  printf("        Nem nehez\n\tC nyelven\n       programozni!\n");
  getchar();    /* az <Enter> leütésére vár */
  return 0;     /* visszatérés 0 értékkel   */
}
```

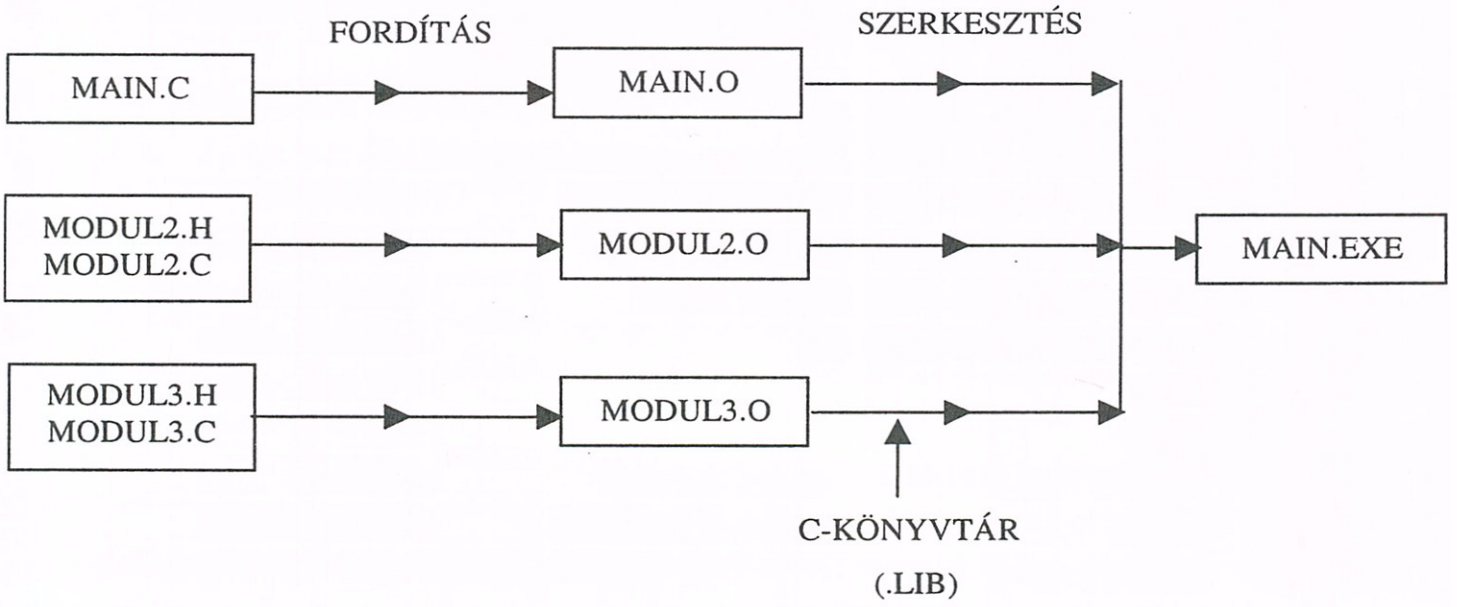
A program futásának eredménye:

```
        Nem nehez
        C nyelven
       programozni!
```

1.3 Több modulból álló C program

A C nyelv támogatja a moduláris programozás elvét, azaz, hogy egy program C kódja több forrásállományban lehessen. A moduláris programozás előnye, hogy a modulok külön is lefordíthatók, így csak azok a modulok fordulnak újra, amelyekben változás történt. Ez a módszer jelentősen csökkenti a program fordításának idejét. A modulokkal érvényesül az adatrejtés elve.

Több modulból álló alkalmazást találunk a könyvünk 7.2. fejezetében.



Megjegyzés: a lefordított tárgymodul kiterjesztése gyakran .OBJ. Amennyiben projektet használunk a modulok összefogására, a futtatható program neve általában megegyezik a projektnévvel.

2. Alaptípusok, változók és konstansok

A C nyelvű programban a használni kívánt neveket mindig deklarálnunk kell, ezáltal adjuk meg a fordítónak a név tulajdonságait. Ha azt szeretnénk, hogy az adott deklarációnak megfelelő memóriafoglalás is végbemenjen, *definíciót* kell használnunk. A *definíció* olyan *deklaráció*, amely helyfoglalással jár. A név definíciója csak egyszer szerepelhet a programban, míg ugyanazt a nevet többször is deklarálhatjuk, de az egymást követő deklarációknak egyezniük kell.

2.1 A C nyelv típusai

A C nyelv alaptípusai táblázatban összefoglalva:

C típusai	*Leírás*
char	karakter,
int	egész szám,
enum	konstansok csoportja,
float	egyszeres pontosságú lebegőpontos szám,
double	kétszeres pontosságú lebegőpontos szám,
struct	struktúra,
union	unió.

Az alaptípusokhoz tartozó **char**, **int** és **double** típuselőírásokhoz bizonyos más kulcsszavakat (típusmódosítókat) kapcsolva, újabb típuselőírásokhoz jutunk, amelynek értelmezése eltér a kiindulási előírástól.

- *Típusmódosítók*: **short**, **long** – a tárolási hosszat, **singed**, **unsigned** – az előjel értelmezését szabályozzák.
- *Típusminősítők*: **const** (csak olvasható), **volatile** (másik futó folyamat is megváltoztathatja).

2.2 Szabványos input és output

Mint már említettük a ***printf()*** kiíró és a ***scanf()*** beolvasó függvénnyel megadott formátum szerint írhatjuk ki, illetve olvashatjuk be az alaptípusú adatokat és karaktersorozatokat (sztringeket). A függvények prototípusa az *STDIO.H* állományban található, melyet a programba a *#include* előfordító utasítással szerkeszthetünk be.

2.2.1 A printf () függvény

A ***printf()*** függvény általános formája:

```
printf("formátumsztring",argumentumlista);
```

A formátumsztring a képernyőre minden átalakítás nélkül kiírásra kerülő karaktersorozatokat és az argumentumlistára vonatkozó konverziós előírásokat tartalmaz. A konverziós előírások részeit, és az egyes részek szerepét az alábbi táblázat mutatja.

Jel	*Jelző*	*Mezőszélesség*	*Pontosság*	*Méretmódosító*	*Konverziós betű*
%	- # 0	15	.5	l	d

Jelző	*Jelentése*
-	balra igazítás,
+	a számoknál az előjel (+,-) mindig megjelenik,
szóköz	előjel helyén a - vagy szóköz jelenik meg,
0	a szóköz 0-val töltődik fel az adat előtt,
#	a hexadecimális számok előtt 0x, az oktális számok előtt 0 előtag jelenik meg.

Egy konverziós előírásnak legalább a kezdő % jelet és a konverzió típusára utaló betűt (kódot) tartalmaznia kell, a többi rész el is maradhat.

Az alap konverziós előírásokat az alábbi táblázat tartalmazza.

Kód	*Argumentum típusa*	*Formátum*
`%c`	**int**	egyetlen karakter,
`%d`	**int**	előjeles decimális egész,
`%ld`	**long int**	előjeles hosszú decimális egész,
`%f`	**float**	[-]ddd.dddd, ahol a tizedesjegyek száma az előírt pontosságtól függ. Fixpontos kiírási mód, alapértelmezés 6 tizedes, 0 pontosság esetén a tizedespont nem jelenik meg,
`%lf`	**double**	hasonló a %f formátumhoz,
`%e`, `%E`	**double**	[-]d.ddddе±ddd, vagy [-]d.ddddE±ddd a tizedesjegyek száma az előírt pontosságtól függ,
`%g`, `%G`	**double**	%e, %E, ha az exponens kisebb, mint -4, vagy >= mint a pontosság, különben %f formátum. Nincsenek vezető nullák és szóközök,
`%o`	**int**	előjel nélküli oktális egész 0 előtag nélkül,
`%s`	**char***	sztring karakterei '\0' (EOS) karakterig,
`%x`, `%X`	**int**	hexadecimális egész 0x előtag nélkül,
`%p`	**void***	mutató értéke.

Írjunk programot, amely karaktert és karaktersorozatot (sztringet) ír ki különféle mezőszélességgel! (*Kiir1*)

```
#include <stdio.h>
int main()
{ char  c  = '*';
  char  *s = "csillag";
  printf("1234567890\n");
  printf("----------\n");
  printf("%c\n",c);
  printf("%4c\n",c);
  printf("%-4c\n",c);
  printf("%s\n",s);
  printf("%10.7s\n",s);
  printf("%10.2s\n",s);
  printf("%-10.7s\n",s);
  printf("%-10.2s\n",s);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
1234567890
----------
*
'   *
*
csillag
   csillag
        cs
csillag
cs
```

Írjunk programot, amely egész típusú változóknak ad értéket, és különféle mezőszélességgel írja ki azt! (*Kiir2*)

```
#include <stdio.h>
int main()
{
  short int     i = 15;
  int           j = -1234;
  unsigned int  k = 3;
  long          m = -12000L;
  printf("1234567890\n");
  printf("----------\n");
  printf("%d\n",i);
  printf("%5d\n",i);
  printf("%-5d\n",i);
  printf("%05d\n",i);
  printf("%d\n",j);
  printf("%6d\n",j);
  printf("%d\n",k);
  printf("%3d\n",k);
  printf("%ld\n",m);
  printf("%8ld\n",m);
  printf("\n%d  %u  %ld\n",j,k,m);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
1234567890
----------
15
   15
15
00015
-1234
 -1234
3
  3
-12000
  -12000

-1234  3  -12000
```

Írjunk programot, amely valós típusú változóknak ad értéket, és különféle mezőszélességgel ír ki! (*Kiir3*)

```
#include <stdio.h>
int main()
{ float   x = -12.456;
  double  y = 2.3456789;
  printf("1234567890\n");
  printf("----------\n");
  printf("%f\n",x);
  printf("%10.3f\n",x);
  printf("%-10.3f\n",x);
  printf("%g\n",x);
  printf("%e\n",x);
  printf("%lf\n",y);
  printf("%e\n",y);
  printf("%9.2e\n",y);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
1234567890
----------
-12.456000
   -12.456
-12.456
-12.456
-1.245600e+001
2.345679
2.345679e+000
2.35e+000
```

2.2.2 A scanf() függvény

A szabványos adatbeviteli eszközről a ***scanf()*** függvény karaktereket olvas, a formátumsztring specifikáció szerint értelmezi és konvertálja a beolvasott karaktereket, majd az argumentum által kijelölt memóriaterületre tölti az eredményt.

Konverziós karakter	*Inputadat*	*Argumentum típusa*
d	decimális egész,	**int***
ld		**long ***
i	egész szám, oktális és hexadecimális is lehet,	**int ***
o	oktális szám a 0 előtag nélkül,	**int***
u	előjel nélküli decimális szám,	**unsigned***
x	hexadecimális egész, 0x vagy 0X előtaggal,	**int***
c	karakter beolvasása, a tagoló karakterek is beolvasásra kerülnek,	**char***
S	sztring olvasása tagoló karakterig,	**char***
f,e,g	lebegőpontos számok beolvasása,	**float***
E,G	%lf esetén az argumentum típusa: Az L előtag esetén az argumentum típusa:	**double*** **long double***
p	a ***printf()*** által kiírt formátumú mutató értékét olvassa be,	**void ** **
n	a formátumig beolvasott karakterek számát adja vissza a függvény az argumentummal kijelölt egész (**int**) típusú változóban.	**int ***

A függvény visszatérési értéke a sikeresen beolvasott adatok száma – ez lehetőséget nyújt az ellenőrzésre.

```
scanf("formátumsztring", argumentumlista);
```

A *scanf()* formátumsztring elemei:

- a tagoló (*whitespace*) karakterek, ezeket a függvény figyelmen kívül hagyja,
- % karakterrel kezdődik a konverziós előírás, mely opcionálisan tartalmazhat:
  - * csillag karaktert, hatására a beolvasott szám eldobódik,
  - n mezőszélességet megadó decimális egész számot,
  - h,l,L opcionális méretkijelölő karaktert:
    h (**short**), l (**long** vagy **double**), L (**long double**).

Írjunk programot, amely egész, valós, karakter és karaktersorozat típusú változóknak billentyűzetről ad értéket, és megjeleníti a beolvasott adatokat! (*Olvas1*)

```
#include <stdio.h>
int main()
{ int    i;
  float  x;
  double y;
  char   c1,c2,c3;
  char   s[11];
  printf("Olvasas klaviaturarol\n");
  printf("   c1 (karakter): ");
  scanf("%c",&c1); /*a c1 változó címére kerül az adat */
  printf("c1 = %c %d\n",c1,c1);
  printf("   i (int): ");
  scanf("%d",&i); /*az i változó címére kerül az adat */
  printf("i = %d\n",i);
  printf("   s(max.10 karakter): ");
  scanf("%s",s); /*az s értéke cím, amely a 0. karakterre mutat */
  printf("s = %s \n",s);
  printf("   x (float): ");
  scanf("%f",&x); /*az x változó címére kerül az adat */
  printf("x=  %f  %e\n",x,x);
  printf("   y (double): ");
  scanf("%lf",&y); /*az y változó címére kerül az adat */
  printf("y= %lf  %e  %g\n",y,y,y);
  fflush(stdin); /* inputpuffer törlése */
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
Olvasas klaviaturarol
   c1 (karakter): a
c1 = a 97
   i (int): 98
i = 98
   s(max.10 karakter): szoveg
```

```
s = szoveg
   x (float): 123.45678
x=  123.456779  1.234568e+002
   y (double): 3.56273e4
y= 35627.300000  3.562730e+004  35627.3
```

2.2.3 A getchar() és a putchar() függvények

Egyetlen karakter beolvasása a billentyűzetről (*stdin*) és kiírása a képernyőre (*stdout*). Erre szolgál két makró, amelyeknek a definíciója az *STDIO.H* fejállományban található:

```
#define getchar(c) getc(stdin)
#define putchar(c) putc((c),stdout)
```

Írjunk programot, amely a ***getchar()*** függvénnyel olvas egy karaktert, és a ***putchar()*** függvénnyel jeleníti meg! (*Olvas2*)

```
#include <stdio.h>
int main()
{
  int ch;
  printf("Karakter: "); ch = getchar();
  printf("A beolvasott karakter: ");
  putchar(ch);
  fflush(stdin);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
Karakter: r
A beolvasott karakter: r
```

2.2.4 A gets() és a puts() függvények

A ***gets()*** függvény egy sort olvas a szabványos inputról. Az *<Enter>* billentyű leütéséig ('\n' karakter az *stdin* adatfolyamban) a karaktersorozatban üres hely (*space*) is előfordulhat.

```
char* gets(char *sptr);
```

A beolvasott sztringet a megadott mutató által kijelölt területre másolja. Mivel a beolvasott sztringben az új sor ('\n') karakter az '\0' (*EOS*) karakterrel helyettesítődik, ezért ez a függvény nem alkalmas újsor karakter beolvasására. Sikeres beolvasás esetén a függvény visszatérési értéke a megadott puffer címe, hiba esetén pedig a *NULL* pointerrel tér vissza.

A ***puts()*** függvény az argumentumban megadott sztringet a szabványos kimenetre (tipikusan a képernyőre) írja, majd a szöveg kiírása után sort emel. Hiba esetén *EOF* (-1) értékkel tér vissza, különben a visszatérési értéke nem negatív.

```
int puts(char *sptr);
```

A ***scanf()*** függvény nem alkalmas olyan karaktersorozat beolvasására, amely szóközt tartalmaz, a ***gets()*** függvény viszont erre a célra kiválóan alkalmas.

Írjunk programot, amely szóközt tartalmazó karaktersorozatot olvas be, és jeleníti, meg! (*Olvas3*)

```
#include <stdio.h>
int main()
{ char str[80];
  fflush(stdin); /*érdemes az inputpuffert törölni*/
  printf("Szoveg olvasasa: "); gets(str);
  printf("Megjelenitese  : "); puts(str);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
Szoveg olvasasa: szovegben szokoz is lehet
Megjelenitese  : szovegben szokoz is lehet
```

Írjunk programot, amely beolvas egy karaktert és egy egész számot, írjuk ki még decimális, oktális, hexadecimális formában is! (*Olvas4*)

```
#include <stdio.h>
int main()
{ int i;  char ch;
  printf("karakter: "); scanf("%c",&ch);
  printf("ch: karakter: %c  decimalis: %d  oktalis: %o  hexa: %x\n",
          ch,ch,ch,ch);
  printf("\negesz szam: "); scanf("%d",&i);
  printf("i : karakter: %c  decimalis: %d  oktalis: %o  hexa: %x\n",
         i,i,i,i);
  printf("\nKiiras 6 mezoszelesseggel\n");
  printf("karakter : %6c\n",i);
  printf("decimalis: %6d\n",i);
  printf("oktalis  : %06o\n",i);
  printf("hexa     : %06x\n",i);
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
karakter: a
ch: karakter: a  decimalis: 97  oktalis: 141  hexa: 61

egesz szam: 98
i : karakter: b  decimalis: 98  oktalis: 142  hexa: 62

Kiiras 6 mezoszelesseggel
karakter :      b
decimalis:     98
oktalis  : 000142
hexa     : 000062
```

2.3 A C nyelv típusainak bemutatása

Az alábbiakban röviden áttekintjük a C nyelv típusait.

2.3.1 Típuselőírások, típusmódosítók

Definiáljunk különböző típusú változókat, használjuk a **short** és a **long** tárolási hossz-, valamint az **unsigned** előjelmódosítót! A definícióban adjunk kezdőértéket, majd írassuk ki a változók tartalmát szöveges formában! (*Tipus*)

```
#include <stdio.h>
int main()
{ short int i = 32000;
  unsigned short j = 65000;
  char c1 = 98;
  unsigned char c2 = 99;
  unsigned long h = 123456789;
  printf(" i  (32000)      = %d\n",i);
  printf(" j  (65000)      = %d\n",j);
  printf(" c1 (98)         = %c\n",c1);
  printf(" c2 (99)         = %c\n",c2);
  printf(" h  (123456789) = %ld\n",h);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
 i  (32000)      = 32000
 j  (65000)      = 65000
 c1 (98)         = b
 c2 (99)         = c
 h  (123456789) = 123456789
```

2.3.2 A felsorolt típus (enum)

Az **enum** típust a konstansnevek felhasználásával hozzuk létre:

enum *azonosító* {*felsorolás*};

Deklaráljunk **enum** típusú konstansokat, állítsunk be az elemeknek különböző értéket! Definiáljunk változókat, amelyek bemutatják az **enum** konstansok használatát! Írassuk ki a változók értékét szöveges formában! (*Enum*)

```
#include <stdio.h>
enum irodalom{vers, oda, regeny, sorozat};
enum utazas{busz, vonat = 4, repulo=15};
```

```
int main()
{
  enum irodalom r1, r2;
  enum utazas mivel1, mivel2;
  r1 = vers; r2 = regeny;
  printf("Irodalom - vers   : %d\n",(int)r1);
  printf("Irodalom - regeny: %d\n",(int)r2);
  mivel1 = busz;
  mivel2 = repulo;
  printf("Utazas - busz     : %d\n",(int)mivel1);
  printf("Utazas - repulo   : %d\n",(int)mivel2);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
Irodalom - vers   : 0
Irodalom - regeny: 2
Utazas - busz     : 0
Utazas - repulo   : 15
```

2.3.3 Saját típusok előállítása

A **typedef** speciális tárolási osztály teszi lehetővé, hogy létező típusokhoz szinonim neveket rendeljünk, és ezt követően már a szinonim névvel is definiálhatunk változókat.

typedef *létező_típus saját_típus*;

A **typedef** használatával rendeljünk szinonim neveket az **int**, **double** és a **char** típusokhoz! A deklarált változóknak adjunk kezdőértéket, és írassuk ki a tartalmukat! (*SajatTipus*)

```
#include <stdio.h>
typedef int egesz;
typedef double valos;
typedef char karakter;
int main()
{
  egesz i = 1973;
  valos x = -4.56;
  karakter c = '@';
  printf("i : %d\n",i);
  printf("x : %lf\n",x);
  printf("c : %c\n",c);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
i : 1973
x : -4.560000
c : @
```

2.3.4 Konstansok a C nyelvben

Definiáljunk konstansokat a **const** használatával, adjunk nekik értéket, melyet felhasználunk a programban! Számítsuk ki a kör kerületét, és a kerület többszörösét a *darab* konstans használatával! (*Konstans*)

Megjegyzés:
A konstans tárolók csak olvashatók a programban, értéküket közvetlenül nem lehet megváltoztatni.

```
#include <stdio.h>
const double PI = 3.141592654;
const int darab = 5;
int main()
{
  double sugar, kerulet, keruletek;
  printf("sugar: "); scanf("%lf",&sugar);
  kerulet = 2 * sugar * PI;
  printf("kerulet  : %lf\n",kerulet);
  keruletek = darab * kerulet;
  printf("keruletek: %lf\n",keruletek);
  fflush(stdin); getchar();
  return 0;
}
```

A program futásának eredménye:

```
sugar: 1
kerulet  : 6.283185
keruletek: 31.415927
```

2.4 Értékek, címek és mutatók

A változók általában az értékadás során kapnak értéket, melynek általános alakja:

objektum = érték;

Az értékadás operátorának (=) bal és jobb oldalán egyaránt szerepelhetnek kifejezések, melyek azonban különböznek egymástól. A bal oldalon szereplő kifejezés azt az *objektumot* jelöli ki a memóriában (megcímezi), ahová a jobb oldalon megadott kifejezés *értékét* be kell tölteni.

2.4.1 Balérték és jobbérték

A C nyelv külön nevet ad az értékadásnál szereplő kétfajta kifejezésnek. Annak a kifejezésnek az értéke, amely az egyenlőség bal oldalán áll, a balérték (*lvalue*), míg a jobb oldalon szereplő kifejezés értéke a jobbérték (*rvalue*).

Példaként nézzünk meg kétfajta értékadást:

```
int x;
x = 7;
x = x + 5;
```

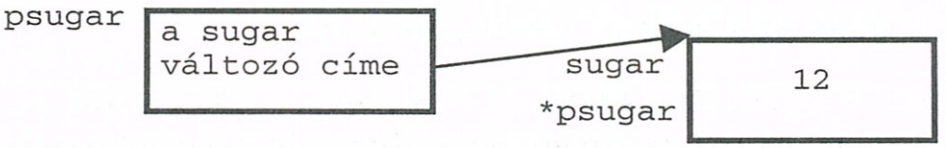
`x = 7;` az *x* változó címe jelöli ki azt az objektumot, ahová a jobb oldalon megadott konstans értékét (7) be kell másolni.

`x = x + 5;` az értékadás során az *x* változó az értékadás mindkét oldalán szerepel. A bal oldalon álló *x* változó címe jelöli ki azt az objektumot, ahová a fordító behelyezi a jobb oldalon álló kifejezés értékét. Vagyis az értékadás előtt az *x* pillanatnyi értékét (7) a konstans 5 értékkel összeadja (elvégzi az összeadást), majd az eredményt az *x*-ben tárolja. Ezáltal az *x* korábbi 7 értéke felülíródik 12-vel.

2.4.2 Ismerkedés a mutatókkal

A C nyelvben kiemelt szereppel rendelkezik egy olyan speciális változó, amely más objektumok címét tárolja – ez a mutató (pointer). Ezzel kapcsolatban az értékadás bal oldalán szerepeltethetünk olyan kifejezéseket is, amelyek balértékek, vagyis cím jellegű kifejezések.

Írjunk programot, amelyben **int** típusú *sugar* címét átveszi az **int*** típusú *psugar* mutató! Adjunk értéket a mutatónak, majd jelenítsük meg a címeket és az adatokat! Változtassuk meg a *sugar* változó értékét, figyeljük meg mi történik! (*MutRef*)



sugar	változó tartalma : 12	*&sugar*	a *sugar* változó memóriabeli címe (& a címe operátor)
psugar	tartalma cím, amely most megegyezik a *sugar* címével	**psugar*	12 (a *psugar* által mutatott változó tartalma.)

```
#include <stdio.h>
int main()
{
  int sugar, *psugar;
  psugar = &sugar;
  *psugar = 12;
  printf("A mutato letezo objektumra mutat\n\n");
  printf("sugar valtozo cime (&sugar)  : %x\n",&sugar);
  printf("psugar mutato tartalma (cim) : %x\n",psugar);
  printf("*psugar (a psugar altal mutatott) valtozo tartalma: %d\n",
         *psugar);
  printf("sugar: %d\n",sugar);
```

```
  printf("\nsugar uj erteke: "); scanf("%d",&sugar);
  printf("*psugar uj erteke: %d azonos a sugar ertekevel: %d\n",
         *psugar,sugar);
  fflush(stdin);getch();
  return 0;
}
```

Megjegyzés: Mivel a *sugar* és a **psugar* ugyanarra a memóriaterületre hivatkozik, akár a **psugar*, akár a *sugar* tartalmát változtatjuk, *mindkettő értéke azonos lesz*! Értékeljük ki a futási eredményt!

A program futásának eredménye:

```
A mutato letezo objektumra mutat

sugar valtozo cime (&sugar)  : 73fdf4
psugar mutato tartalma (cim) : 73fdf4
*psugar (a psugar altal mutatott) valtozo tartalma: 12
sugar: 12

sugar uj erteke: 24
*psugar uj erteke: 24 azonos a sugar ertekevel: 24
```

2.4.3 Mutatók és a dinamikus memóriahasználat

A C nyelv fontos lehetősége az ún. dinamikus memóriahasználat. A program futása során memóriablokkot foglalhatunk le. A sikeres memóriafoglalás eredményeként megkapjuk a lefoglalt blokk címét. A blokkot ezen cím, mint mutató segítségével érhetjük el. Ha már nincs szükségünk a lefoglalt területre, felszabadíthatjuk azt. A felszabadult memóriaterületet a program a futása során újra kioszthatja.

Ha egy mutató értéke a *NULL* konstans, akkor a mutató sehova sem mutat, azaz például jelezhetjük, hogy még számára nem történt helyfoglalás.

A memóriaterület lefoglalását a ***malloc()*** függvénnyel végezhetjük el, melynek argumentumként a lefoglalandó terület nagyságát bájtban kell megadni.

void * ***malloc***(**size_t** *méret*);

A ***malloc()*** függvény használatához az *STDLIB.H* fejállományt kell a programhoz szerkeszteni a *#include* előfordító utasítással.

Olyan mutató használata, amely sehová sem mutat, mert például használata előtt nem előzte meg memóriaterület lefoglalása, a program lefagyásához is vezethet.

A ***malloc()*** függvény visszatérési értékeként kapott, **void*** típusú általános mutatót a megfelelő mutató típusra át kell alakítani. Ha *NULL* értéket kapunk a helyfoglalás eredményeként, akkor valami hiba folytán a helyfoglalás nem történt meg.

Ha már nem szükséges, a mutató számára lefoglalt memóriaterületet, akkor azt a ***free()*** függvénnyel szabadítjuk fel. A ***free()*** hívása után célszerű a mutató értékét *NULL*-ra állítani, ezzel jelezve, hogy már nem mutat sehová.

Írjunk programot, amelyben **int*** típusú mutató számára foglalunk memóriaterületet, műveletvégzés után megjelenítjük az eredményt, majd felszabadítjuk a számára lefoglalt memóriaterületet! (*DimMut*)

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int *ip, j = 2;
  ip = (int*)malloc(sizeof(ip));
  if (ip == NULL)
  {
     printf("Memoriafoglalas nem sikerult!\n");
     getchar(); return 1;
  }
  *ip = j + 6;
  printf("*ip : %d\n",*ip);
  free(ip);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
*ip : 8
```

2.4.4 A void* típusú általános mutató

Írjunk programot, amelyben **void*** típusú általános mutatóval végzünk értékadást, és számolást! (*VoidTip*)

Megjegyzés: A C nyelv lehetővé teszi a típus nélküli, ún. általános mutatók használatát is, azonban ezekkel közvetlenül nem érhető el a mutatott tárterület. A hivatkozott tárolónak csak akkor adhatunk értéket (vagy végezhetünk vele műveletet), ha felhasználói típuskonverzióval (*cast*) típust rendelünk a cím mellé.

```
#include <stdio.h>
int main()
{ int oldal = 3, negyzet_kerulete;
  void* p = &oldal;
  printf("oldal : %d\n",oldal);
  printf("*(int *)p: %d\n", *(int *)p);
  negyzet_kerulete = *(int *)p * 4;
  printf("negyzet kerulete: %d\n",negyzet_kerulete);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
oldal : 3
*(int *)p: 3
negyzet kerulete: 12
```

2.4.5 Többszörös indirektségű mutatók

Írjunk programot, amelyben többszörös indirektségű mutatót használunk, és műveletet végzünk vele! (*TMutato*)

```
#include <stdio.h>
int main()
{
  int k, osszeg;      /* egész változók */
  int *pk;            /* pk egész (int)típusú mutató, azaz
                         int változóra mutathat */
  int **ppk;          /* ppk egy int* típusú mutató, amely
                         int* változóra, vagyis egészre mutató
                         pointerre mutathat  */
  k = 5;
  pk = &k;
  ppk = &pk;
  osszeg = k + *pk + **ppk;
  printf("k      : %d\n",k);
  printf("*pk    : %d\n",*pk);
  printf("**ppk : %d\n",**ppk);
  printf("osszeg: %d\n",osszeg);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
k      : 5
*pk    : 5
**ppk : 5
osszeg: 15
```

Megjegyzés: Mivel a címek azonos helyre mutatnak, az összeg a *k* változó tartalmának háromszorosa lesz.

2.5 Matematikai függvények

A matematikai függvények használatához a *MATH.H* fejállományt kell beszerkeszteni a programba a *#include* előfordító utasítással.

Néhány, gyakran használt matematikai függvény:

gyökvonás	:	`y = sqrt(x);` vagy `y = pow(x,1.0/2);`
koszinusz	:	`y = cos(x);`
szinusz	:	`y = sin(x);`

hatványozás	: a^n	`y = pow(a,n);` vagy `y = exp(log(x)*n));`
exp függvény	: e^x	`y = exp(x);`
természetes log.	: ln x	`y = log(x);`

Írjunk programot, amelyben néhány fontos matematikai függvényt használunk. (*MatFuggv*)

```
#include <stdio.h>
#include <math.h>
const double PI = 3.141592654;
int main()
{
  double adat = 4;
  double gyok, negyzet, kobgyok1, kobgyok2;
  double a = 3, b = 4, szog = 90, c;
  gyok = sqrt(adat);              /* gyökvonás */
  negyzet = pow(adat,2);          /* négyzetre emelés */
  kobgyok1= exp(log(adat)/3.0);   /* köbgyök */
  kobgyok2 = pow(adat, 1.0/3.0); /* köbgyök */
  printf("Adat: %lf\n",adat);
  printf("\nGyok: %lf\n", gyok);
  printf("Negyzet: %lf\n",negyzet);
  printf("1. Kobgyok: %lf\n",kobgyok1);
  printf("2. Kobgyok: %lf\n",kobgyok2);
  c = sqrt( pow(a,2) + pow(b,2)- 2*a*b*cos(szog*PI/180));
  printf("c oldal: %lf\n",c);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
Adat: 4.000000

Gyok: 2.000000
Negyzet: 16.000000
1. Kobgyok: 1.587401
2. Kobgyok: 1.587401
c oldal: 5.000000
```

3. Operátorok és kifejezések

A C nyelvben a legnépesebb utasításcsoportot a pontosvesszővel lezárt kifejezések alkotják. A kifejezések egyetlen operandusból, vagy operandusok és műveleti jelek (operátorok) kombinációjából épülnek fel.

Az operandusok a C nyelv azon elemei, amelyeken az operátorok fejtik ki a hatásukat. Az operandus lehet konstans érték, azonosító, sztring, függvényhívás, tömbindex kifejezés, tagkiválasztó kifejezés (ezek az ún. elsődleges kifejezések) és tetszőleges összetett kifejezés, amely zárójelezett vagy zárójel nélküli, operátorokkal összekapcsolt további operandusokból áll. Ezeket az operátorokat elsődleges (*primary*) kifejezéseknek nevezzük.

Az operátorokat több szempont alapján lehet csoportosítani. Az egyoperandusú (*unary*) operátorok esetén a kifejezés általános alakja:

```
op operandus    vagy    operandus op
```

Az első esetben, amikor az operátor (*op*) megelőzi az operandust, előrevetett (*prefix*), míg a második esetben hátravetett (*postfix*) alakról beszélünk.

`int i;`	egész típusú *i* deklarálva,
`i = 5;`	értékadás,
`i++;`	*i* értékének növelése (*postfix*),
`--i;`	*i* értékének csökkentése (*prefix*),
`(double)i`	*i* értékének valóssá alakítása.

Az operátorok többsége két operandussal rendelkezik, ezek a kétoperandusú (*binary*) operátorok:

```
operandus1 op operandus2
```

Például: `i + 2, i <<3, i+=3` stb.

3.1 Precedencia és asszociativitás

A bonyolultabb kifejezések az elsőbbségi (*precedencia*) szabályok szerint kerülnek kiértékelésre. Ezek a szabályok határozzák meg a kifejezésekben szereplő műveletek kiértékelési sorrendjét.

Egyes operátorok közötti elsőbbségi kapcsolatot táblázatban foglaljuk össze, ahol a táblázat sorai az azonos precedenciával rendelkező operátorokat tartalmazzák. A sorok végén külön jeleztük az azonos precedenciájú operátorokat tartalmazó kifejezésben a kiértékelés irányát, amit *asszociativitásnak* nevezünk. A táblázat első sora tartalmazza a legnagyobb precedenciával rendelkező műveleteket.

Operátor	*Asszociativitás*
`()  []  .  ->`	balról jobbra
`! ~ - ++ -- & * (típus) sizeof`	jobbról balra
`*  /  %`	balról jobbra
`+  -`	balról jobbra
`<<  >>`	balról jobbra
`<  <=  >  >=`	balról jobbra
`==  !=`	balról jobbra
`&`	balról jobbra
`^`	balról jobbra
`|`	balról jobbra
`&&`	balról jobbra
`||`	balról jobbra
`?:`	jobbról balra
`=  +=  -=  *=  /=  %=  <<=  >>=  &=  |=  ^=`	jobbról balra
`,`	balról jobbra

3.1.1 A precedencia szabály

Ha egy kifejezésben különböző precedenciájú műveletek szerepelnek, akkor mindig a magasabb precedenciával rendelkező operátort tartalmazó részkifejezés értékelődik ki először (*PREC1*).

A $a+b*c-d*e$ kifejezés kiértékelési sorrendje azonos az $a+(b*c)-(d*e)$ kifejezésével:

```
1. b * c    -> 12
2. d * e    -> 30
3. a + b * c = a + 12   -> 32
4. a + b * c - d * e = 32 - 30 -> 2
```

```
#include <stdio.h>
int main()
{
  int a, b, c, d, e, ered;
  a = 20; b = 3; c = 4; d = 6; e = 5;
  ered = a + b * c - d * e;
  printf("eredmeny: %d\n",ered);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
eredmeny: 2
```

A kiértékelés sorrendje a matematikából ismert zárójelek segítségével megváltoztatható. Megjegyezzük, hogy a C nyelvben csak a kerek zárójel `()` használható, bármilyen mélységű zárójelezést is hajtunk végre (*PREC2.*)

```
(a + b) * (c - d) * e;
```

A kiértékelés:

```
1. (a + b) -> 23
5. (c - d) -> -2
6. 23 * -2 -> -46
7. -46 * 5 -> -230
```

```
#include <stdio.h>
int main()
{
  int a, b, c, d, e, ered;
  a = 20; b = 3; c = 4; d = 6; e = 5;
  ered = (a + b) * (c - d) * e;
  printf("eredmeny: %d\n",ered);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
eredmeny: -230
```

Ha azonos precedenciájú műveletek szerepelnek egy aritmetikai kifejezésben, akkor a balról-jobbra szabály lép életbe.

A példában egyértelmű, hogy először a b * c szorzás kerül végrehajtásra a precedencia szabály miatt. Mi történik a továbbiakban, mivel a * és az / azonos precedenciájú?

```
a + b * c / d * e;
```

A *d* operandus jobb (*) és bal (/) oldalán azonos precedenciájú művelet szerepel, ekkor a balról jobbra szabály alapján a *d* operandus a tőle balra álló művelethez tartozik, azaz az / (osztás) műveleti jelhez. Ezért *b*c* szorzat után először a *d* operandussal való osztás, majd az *e* operandussal való szorzás hajtódik végre. Matematikai képletre átírva ez jobban látszik:

$$a + \frac{b*c}{d} \cdot e$$

Ha azonban a feladat az alábbi képlet programozása, akkor az ezt leíró utasítást kétféleképpen is megadhatjuk (*PREC3*).

$$a + \frac{b*c}{d*e}$$

Tekintsük meg a *PREC3* mintafeladat utasításait és eredményét!
Szabály, ha a nevezőben szorzat áll, azt kétféleképpen programozhatjuk:

1. Zárójelbe tesszük a nevezőt, hogy mindenképpen a szorzattal osszunk:

```
a + b * c / (d * e);
```

2. Az is jó megoldás, ha a szorzat mindkét operandusával osztunk:

```
a + b * c / d / e;
```

```
#include <stdio.h>
int main()
{
  double a, b, c, d, e, ered1, ered2, ered3;
  a = 20; b = 3; c = 4; d = 6; e = 5;
  ered1 = a + b * c / d * e;
  printf("eredmeny: %lf\n",ered1);
  ered2 = a + b * c / (d * e);
  printf("eredmeny: %lf\n",ered2);
  ered3 = a + b * c / d / e;
  printf("eredmeny: %lf\n",ered3);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
eredmeny: 30.000000
eredmeny: 20.400000
eredmeny: 20.400000
```

Az asszociativitás szabály

Az asszociativitás határozza meg, hogy az adott precedenciaszinten található műveleteket balról-jobbra vagy jobbról-balra haladva kell elvégezni.

Az értékadó utasítások csoportjában a kiértékelést jobbról-balra haladva kell elvégezni, ez teszi lehetővé a C nyelvben több változó együttes értékadását:

```
a = b = c = 0;  azonos az  a = (b = (c = 0));
```

3.2 Mellékhatások és a rövidzár

Bizonyos műveletek feldolgozása során a kifejezés értékének megjelenése mellett bizonyos változók is megváltozhatnak. Ezt a jelenséget mellékhatásnak (*side effect*) nevezzük.

A mellékhatások kiértékelésének sorrendjét nem határozza meg a C szabvány, ezért minden olyan megoldást kerüljünk, ahol a kifejezés eredménye a mellékhatások kiértékelésétől függ:

```
x[i] =  b[i] * i++;
```

Ebben a példában a ++ operátor az, aminek a mellékhatása (*i* értéke 1-gyel nő) teszi bizonytalanná az eredményt.

3.3 Értékadó operátorok

Írjunk programot, amely bemutatja az értékadás szabályait! (*Ertekado*)

Megjegyzés:

- `a = ( b = 5)/4 * 2;` kifejezésben az *a* és a *b* is értéket kap.
- A jobbról-balra szabály lehetővé teszi több változó ugyanazon értékre való állítását: `ered1 = ered2 = 2;`
- Ha egy változó a jobb és a bal oldalon is szerepel, akkor a kifejezés tömörebben is felírható úgy, hogy a műveleti jelet az egyenlőség elé tesszük (üres hely nélkül).
  `ered1 = ered1/(b + 2 - d);` → `ered1 /= b + 2 - d;`
  Lehet még: `+=, -=, *=, %=, &=, |=, ^=, <<=, >>=`

```
#include <stdio.h>
int main()
{ double a, b = 2, c = 10, d = 3, ered1, ered2;
  a = ( b = 5)/4 * 2;
  printf("a: %lf  b: %lf\n",a,b);
  ered1 = ered2 = 2;          /* jobbról balra szabály */
  printf("ered1: %lf      ered2: %lf\n",ered1, ered2);
  // azonos a ered1 = ered1/(b + 2 - d) kifejezéssel
  ered1 /= b + 2 - d;
  printf("ered1 muvelet utani eredmenye: %lf\n",ered1);
  // azonos a ered2 = ered2 + (3 * d - 2) kifejezéssel
  ered2 += 3 * d - 2;
  printf("ered2 muvelet utani eredmenye: %lf\n",ered2);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
a: 2.500000  b: 5.000000
ered1: 2.000000      ered2: 2.000000
ered1 muvelet utani eredmenye: 0.500000
ered2 muvelet utani eredmenye: 9.000000
```

3.4 Léptető (inkrementáló/dekrementáló) operátorok

Írjunk programot, amely bemutatja több egész típusú változó 1-gyel való léptetését! (*Lepteto*)

Megjegyzés:

- Egy numerikus változó eggyel való növelése, illetve csökkentése helyett érdemes a léptető operátort használni:
  `i = i+1;` helyett `i++;` vagy `++i;`
  `i = i-1;` helyett `i--;` vagy `--i;`

`++i; --i;` prefixes alakok.
`i++; i--;` postfixes alakok.

- Vigyázzunk, ha a *prefix* operátort kifejezésben használjuk, akkor a léptetés a kifejezés kiértékelése előtt megy végbe! A postfixes alaknál a léptetés mellékhatásként következik be, tehát például az *i*++ kifejezés kiértékelt értéke *i* eredeti értéke lesz, az *i* változó értéke csak a mellékhatás következtében változik meg.

```
int k = 1, m = 7, n;
n = ++k - --m;  /* k 2, m 6 és n -4 lesz */
```

```
#include <stdio.h>
int main()
{ int i = 2, j = 5, ered1, ered2, ered3, ered4;
  printf("i= %d   j= %d\n",i,j);
  ered1 = i++ + ++j;
  printf("Muvelet vegrehajtasakor j: %d  ered1: %d\n",j,ered1);
  printf("\nMuvelet utan\n");
  printf("i= %d  j= %d\n",i,j);
  ered2 = ++i + j++;
  printf("Muvelet vegrehajtasakor i: %d   ered2: %d\n",i, ered2);
  printf("\nMuvelet utan\n");
  printf("i= %d  j= %d\n",i,j);
  ered3 = i-- + j--;
  printf("ered3: %d\n",ered3);
  printf("\nMuvelet utan\n");
  printf("i= %d  j= %d\n",i,j);
  ered4 = --i + --j;
  printf("Muvelet vegrehajtasakor i: %d   j: %d  ered4: %d\n",
          i,j,ered4);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
i= 2   j= 5
Muvelet   vegrehajtasakor   j:   6
ered1: 8

Muvelet utan
i= 3  j= 6
Muvelet   vegrehajtasakor   i:   4
ered2: 10

Muvelet utan
i= 4  j= 7
ered3: 11

Muvelet utan
i= 3  j= 6
Muvelet vegrehajtasakor i: 2    j:
5  ered4: 7
```

3.5 Bitműveletek

Írjunk programot, amelyben **short int** típusú változókon végzünk *vagy* (|), *kizáró vagy* (^), *és* (&) valamint *tagadás*(~) logikai műveleteket! (*Bitmuv1*)

```
#include <stdio.h>
int main()
{
  short int i = 0x0027, j1 = 0x0012, vagy;
  short int j2 = 0x0006, es;
  short int j3 =0x0013, kizarovagy;
  short int negalas;
  vagy = i | j1;
  printf("%04x | %04x = %04x\n",i,j1,vagy);
  es = i & j2;
  printf("\n%04x & %04x = %04x\n",i,j2,es);
  kizarovagy = i ^ j3;
  printf("\n%04x ^ %04x = %04x\n",i,j3,kizarovagy);
  negalas = ~i;
  printf("\n~%04x = %04x",i,negalas);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
0027 | 0012 = 0037

0027 & 0006 = 0006

0027 ^ 0013 = 0034

~0027 = ffffffd8
```

```
  0000 0000 0010 0111
| 0000 0000 0001 0010
  0000 0000 0011 0111   →  37

  0000 0000 0010 0111
& 0000 0000 0000 0011
  0000 0000 0000 0011   →  6

  0000 0000 0010 0111
^ 0000 0000 0001 0011
  0000 0000 0010 0111   →  34

  0000 0000 0010 0111
~ 1111 1111 1101 1000   →  ffd8

  az előző 4 bájt ffff
```

Írjunk programot, amelyben **short int** típusú értékeken bemutatja a *kizáró vagy* művelettel való kódolást és a kódolt adat visszaállítását! (*Bitmuv2*)

```
#include <stdio.h>
int main()
{
  short int kodolo = 0x0263, adat = 0x1973;
  printf("Adat: %04x  ^  %04x\n",adat,kodolo);
  adat = adat ^ kodolo;
```

```
  printf("Kodolt adat: %04x\n",adat);
  adat = adat ^ kodolo;
  printf("Visszakodolas (^): %04x\n",adat);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
Adat: 1973  ^  0263
Kodolt adat: 1b10
Visszakodolas (^): 1973
```

3.5.1 Biteltoló műveletek

Írjunk programot, amely egy **short int** típusú értéket kettővel jobbra, illetve balra léptet! (*Bitlepteto*)

Megjegyzés:

- Egész szám bitjeinek *n* lépéssel történő balra tolása a szám 2^n értékkel való szorzását eredményezi.
- Egész szám bitjeinek *n* lépéssel történő jobbra tolása a szám 2^n értékkel elvégzett egész osztásának felel meg.

Egy egész szám 2^n-nel való szorzásának/osztásának ez a leggyorsabb módja.

```
#include <stdio.h>
int main()
{
  short int a = 0x0065, ered1, ered2;
  ered1 = a >> 2;
  printf("%x >> 2 = %x\n",a,ered1);
  ered2 = a << 2;
  printf("%x << 2 = %x\n",a,ered2);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
65 >> 2 = 19
65 << 2 = 194
```

0000 0000 0110 0101 >> 2	0000 0000 0001 1001 -> 19
0000 0000 0110 0101 << 2	0000 0001 1001 0100 ->194

3.6 A sizeof operátor

Írjunk programot, amely különböző típusú változóknak kiíratjuk a bájtban kifejezett méretét! (*Meret*)

Megjegyzés:

- A **sizeof** művelet kétféleképpen alkalmazható:
  ```
  meret = sizeof  változó; /* a változó nevével */
  meret = sizeof (int); /* a változó típusával, zárójelben. */
  ```
- **sizeof(char)** definíció szerint 1.

```
#include <stdio.h>
int main()
{
  int i;
  short int j;
  long int k;
  float x;
  double y;
  unsigned char w;
  int m1, m2, m3, m4, m5, m6;
  m1 = sizeof(int);
  m2 = sizeof j;
  m3 = sizeof(long int);
  m4 = sizeof(float);
  m5 = sizeof(double);
  m6 = sizeof w;
  printf("int          merete: %d\n",m1);
  printf("short int    merete: %d\n",m2);
  printf("long int     merete: %d\n",m3);
  printf("float        merete: %d\n",m4);
  printf("double       merete: %d\n",m5);
  printf("unsigned char merete: %d\n",m6);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
int          merete: 4
short int    merete: 2
long int     merete: 4
float        merete: 4
double       merete: 8
unsigned char merete: 1
```

Megjegyzés: a **long double** mérete implementációfüggő!

3.7 A vessző operátor

Írjunk programot, amely értékadásnál, illetve műveletben használja a vessző operátort! (*Vesszo*)

Megjegyzés:

- A vessző operátort tartalmazó kifejezés balról-jobbra haladva értékelődik ki, a kifejezés értéke és típusa megegyezik a jobb oldali operandus értékével, illetve típusával.
- A programban `ered3 = i * (adat = 10, adat + 15);` kifejezésben az *adat* megkapja a *10* értéket, majd ehhez az értékhez hozzáadódik a *15*, és végül megszorzódik az *i* értékével (*12*-vel). Az *ered3* értéke 300 lesz.

```
#include <stdio.h>
int main()
{
  int i = 1, j, k, ered2, ered1, adat = 0, ered3;
  ered1 = ered2 = 0;    /* jobbról-balra szabály */
  i = 12, j = 31;       /* a vessző operátor használata */
  ered3 = i * (adat = 10, adat + 15); /*a vessző operátor használata*/
  printf("adat erteke  : %d\n",adat);
  printf("ered3 muvelet: %d\n",ered3);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
adat erteke  : 10
ered3 muvelet: 300
```

3.8 A feltételes operátor

Írjunk programot, amelyben feltételes operátort használunk két változó közül a legnagyobb értékű kiválasztására! (*Feltelop*)

Megjegyzés:

- A feltételes operátor (?:) három operandussal rendelkezik:
  feltétel_kif ? igaz_kif : hamis_kif
  Ha a *feltétel_kif* nem 0 (igaz), akkor az *igaz_kif* értéke adja a feltételes kifejezés értékét, ellenkező esetben a kettőspont (:) után lévő *hamis_kif*.
- Az eredményhez kétféleképpen is hozzáférhetünk (bemutatja a példa).

```
#include <stdio.h>
int main()
{
  int x = 12, y = 2, min1, min2;
  x < y ? (min1 = x) : (min1 = y); /* zárójelet kell alkalmazni */
  min2 = x < y ? x : y;
  printf("Legkisebb ertek (min1): %d\n",min1);
  printf("Legkisebb ertek (min2): %d\n",min2);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
Legkisebb ertek (min1): 2
Legkisebb ertek (min2): 2
```

3.9 Statikus és dinamikus helyfoglalású változók használata

Írjunk programot, amelyben egész típusú, statikus helyfoglalású változókon végzünk műveleteket! Mit tapasztalunk? (*Muvelet1*)

Megjegyzés:

- Két egész szám hányadosa egész. Ha el szeretnénk kerülni az osztás eredményének automatikus egésszé csonkolását, akkor a számlálót, vagy a nevezőt valóssá kell alakítanunk.
- Vagy az egész számot tizedesponttal, vagy a változót **double** típusmódosítással átalakítjuk valóssá, hogy az osztás művelet elvégzésekor a pontos eredményt kapjuk valós számként:

```
atlag2 = (adat1 + adat2)/2.0
atlag3 = (double)(adat1 + adat2)/2;
```

- Változó beolvasása a ***scanf()*** beolvasó függvénnyel történik. A ***printf()*** függvénnyel rákérdezünk az adatra, és ***scanf()*** segítségével olvassuk be az adatot:

```
printf("1.adat: "); scanf("%d",&adat1);
```

```
#include <stdio.h>
int main()
{
  int adat1, adat2, kulonbseg;
  double osszeg, hanyados, atlag1, atlag2, atlag3;
  printf("1.adat: "); scanf("%d",&adat1);
  printf("2.adat: "); scanf("%d",&adat2);
  osszeg = adat1 + adat2;
  printf("\nosszeg: %lf\n",osszeg);
  kulonbseg = adat1 - adat2;
  printf("\nkulonbseg: %d",kulonbseg);
```

```
  hanyados = adat1/adat2;  /* értékvesztés az egészek osztása miatt */
  printf("\nhanyados: %lf\n",hanyados);
  atlag1 = (adat1 + adat2)/2; /* értékvesztés */
  atlag2 = (adat1 + adat2)/2.0; /* valós típusú osztás */
  atlag3 = (double)(adat1 + adat2)/2;  /* valós típusú osztás */
  printf("\natlag1: %lf\n",atlag1);
  printf("atlag2: %lf\n",atlag2);
  printf("atlag3: %lf\n",atlag3);
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
1.adat: 1
2.adat: 2

osszeg: 3.000000

kulonbseg: -1
hanyados: 0.000000

atlag1: 1.000000
atlag2: 1.500000
atlag3: 1.500000
```

3.9.1 Mutatók használata

Írjunk programot, amelyben valós változókon végzünk műveleteket, az eredményeket pedig mutatók segítségével tároljuk! (*Muvelet2*)

Megjegyzés:

- A dinamikus változónak a ***malloc()*** függvénnyel foglalunk helyet a memóriában, megadva annak típusát: `szk = (double*)malloc(sizeof(double));`
- A ***free()*** függvény felszabadítja a ***malloc()*** függvény által lefoglalt területet:
  `free(szk);`

```
#include <stdio.h>
#include <math.h> /* sqrt() használatához */
int main()
{
  double *adat1 = NULL, *adat2 = NULL, *atlag = NULL;
  adat1 = (double*)malloc(sizeof(double));  /* memóriafoglalás */
  adat2 = (double*)malloc(sizeof(double));  /* memóriafoglalás */
  if (adat1 == NULL || adat2 == NULL)
  {  printf("Memoriafoglalas nem sikerult!\n"); getchar();return 1; }
  printf("adat1: "); scanf("%lf",adat1); /* adat1 címet tartalmaz */
  printf("adat2: "); scanf("%lf",adat2); /* adat2 címet tartalmaz */
  atlag = (double*)malloc(sizeof(double));  /* memóriafoglalás */
  if (atlag == NULL)
  { printf("Memoriafoglalas nem sikerult!\n"); getchar();return 1; }
```

```
  *atlag = (*adat1 + *adat2)/2;
  printf("Adatok atlaga: %lf\n",*atlag);
  free (adat1); /* a lefoglalt memória felszabadítása */
  free (adat2); free (atlag);
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
adat1: 3
adat2: 7
Adatok atlaga: 5.000000
```

3.10 Konstansok használata

Írjunk programot, amelyben a gömb felszínének és térfogatának számítása során a π értékét konstansként használjuk, valamint a kiírandó szöveget mutató segítségével tároljuk! Használjuk a *math.h* fejállományt a ***pow()*** matematikai függvény eléréséhez! (*Konstans*)

```
#include <stdio.h>
#include <math.h>
char *szoveg1 = "A gomb felszine : ";
char *szoveg2 = "A gomb terfogata: ";
const double PI = 3.141592654;
int main()
{
  double sugar, gombfelsz, gombterf;
  printf("Gomb sugara: "); scanf("%lf",&sugar);
  gombfelsz = 4 * pow(sugar, 2)* PI;
  gombterf = 4 * pow(sugar, 3)* PI/3;
  printf("%s %lf\n", szoveg1, gombfelsz);
  printf("%s %lf\n", szoveg2, gombterf);
  fflush(stdin);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
Gomb sugara: 1
A gomb felszine :  12.566371
A gomb terfogata:  4.188790
```

3.11 Saját típus előállítása

Írjunk programot, amelyben a változókat saját típussal definiáljuk! Olvassuk be a henger sugarát és magasságát, számítsuk ki a henger felszínét és térfogatát! A számításához definiáljuk a π értékét konstansként! A hatványozáshoz használjuk a ***pow()*** matematikai függvényt a *math.h* deklarációs állományból! (*Typedef*)

```
#include <stdio.h>
#include <math.h>
const double PI = 3.141592654;
typedef int egesz;
typedef double valos;
int main()
{
  egesz sugar, magassag;
  valos hengfelsz, hengterf;
  printf("Henger sugara: "); scanf("%d",&sugar);
  printf("Henger magassaga: "); scanf("%d",&magassag);
  hengfelsz = 2 * pow(sugar,2) * PI + 2 * sugar * PI * magassag;
  hengterf = pow(sugar, 2) * PI * magassag;
  printf("\nHenger felszine : %lf\n",hengfelsz);
  printf("Henger terfogata: %lf\n",hengterf);
  fflush(stdin);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
Henger sugara: 1
Henger magassaga: 1

Henger felszine : 12.566371
Henger terfogata: 3.141593
```

4. A C nyelv utasításai

Ez a fejezet különféle feladatokat mutat be a C nyelv utasításainak használatára. Az utasításokat többféleképpen is csoportosíthatjuk. A szelekciós utasítások az egyszerű, illetve többszörös elágazást lehetővé tevő **if** és **switch** utasítások jelölik. Az iterációs utasítások, a C nyelv ciklusutasításait (**while**, **for**, **do while**) csoport tartalmazza.

4.1 Utasítások és blokkok

Tetszőleges kifejezés utasítássá válik, ha pontosvesszőt (;) helyezünk mögé – ez a kifejezésutasítás:

```
--k;  i++;
```

Egyetlenegy pontosvesszőből áll az üres utasítás, amely utasítást helyettesít, amennyiben mást nem akarunk végrehajtani azon a helyen.

Kapcsos zárójelekkel `{ }` több utasítást egyetlen utasítássá vonhatunk össze. A szelekciós és iterációs utasítások alaphelyzetben csupán egy tevékenységet (utasítás) hajtanak végre. A kapcsos zárójelekkel összefogott utasítások azonban egy utasításnak látszanak számukra. Ha elfelejtjük blokkba tenni a megfelelő utasításokat, akkor csak az elsőt futtatják.

4.2 Az if utasítás

Feltételes utasításnak két formája van.

- Az **if** utasítás:

  if *(kifejezés)*
  utasítás

  Ha a *kifejezés* nem nulla, vagyis a feltétel igaz, így az *utasítás* végrehajtódik.

- Ha a **if-else** utasításnál a feltétel teljesül, akkor az *utasítás1* kerül végrehajtásra, különben pedig az *utasítás2*.

  if *(kifejezés)*
  utasítás1
  else
  utasítás2

Írjunk programot, amely beolvas egy egész számot, és kijelzi, ha páros, illetve ha páratlan, akkor megnézi a 100-hoz való viszonyát! (*Feltetel1*)

```
#include <stdio.h>
int main()
{ int x;
  printf("Egesz szam: "); scanf("%d",&x);
  if ( x % 2 == 0 ) printf(" paros szam\n");
  else if (x > 100)
   printf(" nagyobb 100-nal\n");
  else
   printf(" nem nagyobb 100-nal\n");
  fflush(stdin); getchar();
  return 0;
}
```

A program futásainak eredménye:

```
Egesz szam: 13
 nem nagyobb 100-nal
Egesz szam: 12
 paros szam
```

Írjunk programot, amely három valós pozitív számnak kiszámítja az átlagát, hibát jelez, ha a kívánt (pozitív) feltétel nem teljesül! (*Feltetel2*)

```
#include <stdio.h>
int main()
{
  double x1, x2, x3, atlag;
  printf("Harom pozitiv szam atlaga\n");
  printf("1. adat : "); scanf("%lf",&x1);
  printf("2. adat : "); scanf("%lf",&x2);
  printf("3. adat : "); scanf("%lf",&x3);
  if (x1 > 0 && x2 > 0 && x3 > 0)
  {
    atlag = (x1 + x2 + x3)/3;
    printf("\nAtlag: %lf\n",atlag);
  }
  else printf("Hibas adat!\n");
  fflush(stdin); getchar();
  return 0;
}
```

A program futásainak eredménye:

```
Harom pozitiv szam atlaga
1. adat : 1
2. adat : 2
3. adat : 3
Atlag: 2.000000

Harom pozitiv szam atlaga
1. adat : 1
2. adat : -2
3. adat : 3
Hibas adat!
```

Írjunk programot, amely beolvas egy pozitív egész számot, megvizsgálja, hogy osztható-e 4-gyel, illetve 6-tal, valamint páratlan és osztható-e 3-mal, vagy egyikkel sem osztható! (*Feltetel3*)

```
#include <stdio.h>
int main()
{
  int szam, db = 0;
  printf("Pozitiv szam: "); scanf("%d",&szam);
  if (szam <= 0) printf("Hibas adat!\n");
  else
  {
   if (szam % 4 == 0)
   {
     printf("oszthato 4-gyel\n");
     db++;
   }
   if (szam %6 == 0)
   {
     printf("oszthato 6-tal\n");
     db++;
   }
   if (szam % 2 == 1 && szam % 3 == 0)
   {
     printf("Paratlan es oszthato 3-mal is\n");
     db++;
   }
   if (!db)
      printf("a 4, a 6 es a 3 a szamnak nem osztoi\n");
  }
  fflush(stdin);
  getchar();
  return 0;
}
```

A program futásainak eredménye:

```
Pozitiv szam: 24
oszthato 4-gyel
oszthato 6-tal

Pozitiv szam: 21
Paratlan es oszthato 3-mal is

Pozitiv szam: 37
a 4, a 6 es a 3 a szamnak nem osztoi
```

Írjunk programot, amely beolvassa egy dolgozat pontszámát, és a ponthatárok alapján leosztályozza azt! (*Feltetel4*)

Ponthatár	*Érdemjegy*
0 - 50	1
51 - 65	2
66 - 75	3
76 - 85	4
86 - 100	5

```
#include <stdio.h>
int main()
{
  int pontszam, erdemjegy;
  printf("Pontszam: "); scanf("%d",&pontszam);
  if (pontszam >= 0 && pontszam <= 100)
  {
    if (pontszam >= 0 && pontszam <= 50) erdemjegy = 1;
    else if (pontszam >= 51 && pontszam <= 65) erdemjegy = 2;
    else if (pontszam >= 66 && pontszam <= 75) erdemjegy = 3;
    else if (pontszam >= 76 && pontszam <= 85) erdemjegy = 4;
    else if (pontszam >= 86 && pontszam <= 100) erdemjegy = 5;
    printf("Erdemjegy: %d\n",erdemjegy);
  }
  else printf("Hibas adat!\n");
  fflush(stdin); getchar();
  return 0;
}
```

A program futásainak eredménye:

```
Pontszam: 78
Erdemjegy: 4

Pontszam: 69
Erdemjegy: 3
```

Írjunk programot, amely beolvassa a víz hőmérsékletét, és a hőmérséklet alapján szövegesen megjeleníti a víz halmazállapotát! (*Feltetel5*)

Algoritmus:

fok <= 0 szilárd (jég), 0 < fok <100 folyékony (víz), fok >= 100 légnemű (gőz).

```
#include <stdio.h>
int main()
{
  int fok;
  printf("Viz homerseklete: ");
  scanf("%d",&fok);
  if (fok <= 0) printf("  szilard (jeg)\n");
```

```
  else if (fok >0 && fok < 100) printf("  folyekony (viz)\n");
  else if (fok <=100) printf("  legnemu (goz)\n");
  fflush(stdin); getchar();
  return 0;
}
```

A program futásainak eredménye:

```
Viz homerseklete: 30
  folyekony (viz)

Viz homerseklete: 0
  szilard (jeg)

Viz homerseklete: 100
  legnemu (goz)
```

Írjunk programot, amely bemutatja a rövidzár kiértékelés működését! (*Rovidzar*)

Megjegyzés:

Ha a feltételes utasításban a *vagy* kapcsolat bal oldala teljesül, akkor a jobb oldalán lévő kifejezés már nem kerül kiértékelésre, azaz nem hajtódik végre. Ugyanazt tapasztaljuk, ha az *és* kapcsolat bal oldala hamis.

```
int logikai_fv(int par);
int a = 1, b = 0, d;

if ( a || logikai_fv(d))


if (b && logikai_fv(d))
```

Az első feltételben az *a* nem nulla, akkor *logikai_fv(d)* függvény nem hívódik meg. A második feltételben *b* nem 1, akkor a *logikai_fv(d)* függvény nem hívódik meg.

4.3 A switch utasítás

A vezérlés egy *kifejezés* értékétől függően több lehetséges utasítás valamelyikére kerül a **switch** típusú elágaztatásban. Az utasítás általános alakja:

switch *(kifejezés)*
{
  case *konstans_kifejezés:*
    utasítások
  . . .
  default*:*
    utasítások
}

- A **switch** utasítás kiértékeli a *kifejezést,* és arra a **case** címkére adja át a vezérlést, amelyben a *konstans_kifejezés* értéke megegyezik a *kifejezés* értékével. A **default** címkével megjelölt utasítással folytatódik a program futása, ha egyikkel sem egyezik meg.
- A konstans kifejezés csak egész jellegű (**int**, **enum**, **char** stb. lehet).
- Az adott címkéhez tartozó programrészlet végrehajtása után a **goto**, **break** vagy **return** utasítással léphetünk ki a **switch** utasításból.

Írjunk programot, amely beolvas két valós számot és egy karaktert, amely meghatározza, hogy szorzatot (s,S), vagy különbséget (k,K) számolunk! Ellenőrizzük az adatok helyességét, és szövegesen jelenítsük meg az eredményt! (*Switch1*)

```
#include <stdio.h>
#include <math.h>
#include <ctype.h> /* a toupper() függvény használatához szükséges */
int main()
{
  double a, b, eredmeny;
  char mod;
  printf("s, S: szorzat szamitasa\n");
  printf("k, K: kulonbseg szamitasa\n");
  printf("1. adat: " ); scanf("%lf",&a);
  printf("2. adat: " ); scanf("%lf",&b);
  printf("Melyik muvelet (s,S vagy k,K): ");
  fflush(stdin); /* input buffer ürítése a karakter olvasás számára */
  scanf("%c",&mod);

  if ( toupper(mod) == 'S' || toupper(mod) == 'K')
  {
    switch (mod)
    {
     case 's':
     case 'S': eredmeny = a * b;
          printf("\nSzorzat = %lf\n",eredmeny); break;
     case 'k':
     case 'M': eredmeny = a - b;
          printf("\nKulonbseg = %lf\n",eredmeny); break;
    }
  }
  else
     printf("Hibas adat!\n");
  fflush(stdin); getchar();
  return 0;
}
```

A program futásainak eredménye:

```
s, S: szorzat szamitasa
k, K: kulonbseg szamitasa
1. adat: 8
2. adat: 6
```

```
Melyik muvelet (s,S vagy k,K): k
Kulonbseg = 2.000000

s, S: szorzat szamitasa
k, K: kulonbseg szamitasa
1. adat: 8
2. adat: 6
Melyik muvelet (s,S vagy k,K): s
Szorzat = 48.000000
```

Írjunk programot, amely beolvassa egy dolgozat pontszámát és a ponthatárok alapján leosztályozza azt! Az osztályzatot számmal és szövegesen egyaránt jelenítsük meg! (*Switch2*)

```
#include <stdio.h>
int main()
{
  int pontszam, erdemjegy;
  printf("Pontszam: "); scanf("%d",&pontszam);
  if (pontszam >= 0 && pontszam <= 100)
  {
    if (pontszam >= 86) erdemjegy = 5;
    else if (pontszam >= 76) erdemjegy = 4;
    else if (pontszam >= 66) erdemjegy = 3;
    else if (pontszam >= 51) erdemjegy = 2;
    else erdemjegy = 1;

    printf("Erdemjegy: %d",erdemjegy);
    switch (erdemjegy)
    {
      case 1:  printf(" (elegtelen) \n"); break;
      case 2:  printf(" (elegseges) \n"); break;
      case 3:  printf(" (kozepes) \n"); break;
      case 4:  printf(" (jo) \n"); break;
      case 5:  printf(" (jeles) \n"); break;
    }
  }
  else printf("Hibas adat!\n");
  fflush(stdin); getchar();
  return 0;
}
```

A program futásainak eredménye:

```
Pontszam: 67
Erdemjegy: 3 (kozepes)

Pontszam: 89
Erdemjegy: 5 (jeles)
```

Írjunk programot, amely a beolvas két valós adatot, valamint egy műveleti jelet (+,-,*,/)! A műveleti jel alapján számítsuk ki az eredményt adjunk hibajelzést, ha hibás a műveleti jel, vagy osztás esetén a nevező zérus! (*Switch3*)

Megjegyzés: Ha a műveleti jel / (osztás), akkor a nevező értékét lebegőpontosan tíz karakter mezőszélességgel (%10e), más műveleti jel esetén fixpontosan 8 mezőszélességgel két tizedes (%8.2lf) jegyre írjuk ki.

```
#include <stdio.h>
#include <math.h>
int main()
{
  char muvjel;
  double adat1, adat2, ered;
  printf("1. adat: "); scanf("%lf",&adat1);
  printf("2. adat: "); scanf("%lf",&adat2);
  fflush(stdin); /* input buffer törlése karakter beolvasásához */
  printf("Muveleti jel (+,-,*,/): "); scanf("%c",&muvjel);

  switch (muvjel)
  {
    case '+': ered = adat1 + adat2; break;
    case '-': ered = adat1 - adat2; break;
    case '*': ered = adat1 * adat2; break;
    case '/': if (fabs(adat2) <1.0e-6) /* ha a nevező abszolút értéke
                  kisebb, mint 0.000001, akkor zérusnak tekintjük */
              {
                 printf("Nevezo zerus\n");
                 getchar();
                 return 1;
              }
              else
              {
                ered = adat1 / adat2; break;
              }
    default:
        {
          printf("Hibas muveleti jel!\n"); getchar();
          return 1;
        }
  }
  if (muvjel == '/')
    printf("\n%8.2lf %2c %10e = %8.2lf\n",adat1,muvjel,adat2,ered);
  else
    printf("\n%8.2lf %2c %8.2lf = %8.2lf\n",adat1,muvjel,adat2,ered);

  fflush(stdin);getchar();
  return 0;
}
```

A program futásainak eredménye:

```
1. adat: 3
2. adat: 1.5e-7
Muveleti jel (+,-,*,/): /
Nevezo zerus

1. adat: 3
2. adat: 0.0015
Muveleti jel (+,-,*,/): /

    3.00  / 1.500000e-003 =  2000.00

1. adat: 4.5
2. adat: 2
Muveleti jel (+,-,*,/): *

    4.50  *     2.00 =     9.00
```

4.4 Ciklusutasítások

A programozási nyelvekben bizonyos utasítások automatikus ismétlését biztosító programszerkezetet iterációnak vagy ciklusnak (*loop*) nevezzük. Ez az ismétlés mindaddig tart, amíg az ismétlési feltétel igaznak bizonyul.

4.4.1 A while ciklus

A **while** ciklus mindaddig ismétli a ciklus törzsében lévő utasításokat, míg a vizsgált *kifejezés* (vezérlőfeltétel) értéke igaz (nem 0).

Az utasítás általános alakja:

while (*kifejezés*)
{
  /* *a ciklus törzse* */
}

Ez egy ún. elől tesztelt ciklus. Nem biztos, hogy a ciklusmag egyáltalán lefut.

Írjunk programot, amely beolvassa az adatok darabszámát, majd kiszámítja a páratlanok szorzatát és a páros adatok összegét! (*While1*)

```
#include <stdio.h>
int main()
{
  int i, n;
  int osszeg = 0, szorzat = 1;
  i = 0;
```

```
    while (i < 7)
    {
      if ( i %2 == 0) osszeg += i;
      else szorzat *= i;
      i++;
    }
    printf("Paros szamok osszege     : %d\n",osszeg);
    printf("Paratlan szamok szorzata: %d\n",szorzat);
    getchar();
    return 0;
}
```

A program futásának eredménye:

```
Paros szamok osszege     : 12
Paratlan szamok szorzata: 15
```

Írjunk programot, amely pontig (.) karaktereket olvas! Számláljuk meg az *a*, *A*, *e*, a szóköz az egyéb karaktereket, valamint az összes beolvasott karaktert a pont nélkül! (*While2*)

```
#include <stdio.h>
int main()
{
  int ch, egyeb = 0, a_kar = 0, A_kar = 0, e_kar = 0,
      szokoz = 0, osszes_db = 0;
  printf("Szoveg (. zarja): ");
  while ((ch = getchar()) != '.')
  {
    osszes_db++;
    switch(ch)
    {
      case 'a': a_kar++; break;
      case 'A': A_kar++; break;
      case 'e': e_kar++; break;
      case ' ': szokoz++; break;
      default: egyeb++;
    }
  }
  printf("a karakterek szama: %d\n",a_kar);
  printf("A karakterek szama: %d\n",A_kar);
  printf("e karakterek szama: %d\n",e_kar);
  printf("szokoz karakterek szama: %d\n",szokoz);
  printf("egyeb karakterek szama : %d\n",egyeb);
  printf("osszes karakterek szama: %d\n",osszes_db);
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
Szoveg (. zarja): A mai nap meleg lesz.
a karakterek szama: 2
A karakterek szama: 1
e karakterek szama: 3
szokoz karakterek szama: 4
egyeb karakterek szama : 10
osszes karakterek szama: 20
```

Írjunk programot, amely végtelen **while** ciklusban a nulla érték megadásáig olvas egész számokat! Páratlan és hárommal osztható szám beolvasásakor **continue** utasítással hagyjuk ki az aktuális ciklusléptést! Határozzuk meg a páros adatok számát, valamint a beolvasott összes adat számát. Számítsuk ki a beolvasott adatok összegét, a páratlan és a hárommal oszthatók kivételével! (*While3*)

Megjegyzés:

- A páratlan és hárommal osztható szám beolvasásakor a **continue** utasítás a következő iterációt hajtja végre, a **continue** mögötti utasítások ekkor kimaradnak.
- A páros számokat számláljuk.
- Ha a beolvasott adat 0, akkor a **break** utasítással lépjünk ki a végtelen ciklusból!

```
#include <stdio.h>
int main()
{
  int i, n = 0, adat, osszeg = 0, paros = 0;
  while(1)
  {
    printf("Adat: "); scanf("%d",&adat);
    n++; /* a beolvasott adatok száma */
    if (adat %2 != 0 && adat % 3 == 0)
    {
      printf("Paratlan es 3-mal oszthato: %d -> continue\n",adat);
      continue;
    }
    if (adat %2 == 0) paros++; /*páros adatok számlálása */
    if (adat == 0)
   {
      printf("zerus adat -> break\n");
      break;  /* kilépés a végtelen ciklusból */
   }
    osszeg += adat;
  }
  printf("Adatok szama: %d\n",n);
  printf("Parosak szama: %d\n",paros);
  printf("Osszeg: %d\n",osszeg);
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
Adat: 2
Adat: 6
Adat: 3
Paratlan es 3-mal oszthato: 3 -> continue
Adat: 4
Adat: 21
Paratlan es 3-mal oszthato: 21 -> continue
Adat: 7
Adat: 0
zerus adat -> break
Osszes beolvasott adat darabszama: 7
Parosak szama: 4
Osszeg: 19
```

4.4.2 A for ciklus

A **while** ciklus használatának az egyik legjellemzőbb kódja ez:

inicializáló_kif;
while (*feltétel_kif*)
{
  lépésköz_kif;
}

Ennek rövidített leírására szolgál a **for** ciklus.

A **for** utasítást általában akkor használjuk, ha a ciklus törzsében megadott utasításokat egy számsorozat elemein végig futtatva kívánjuk végrehajtani. A **for** utasítás általános formája:

for(*inicializáló_kif*; *feltétel_kif*; *léptető_kif*)
{
  /* *a ciklus törzse* */
}

Írjunk programot, amely a 0-tól 6-ig futó között működő **for** ciklusban kiszámítja a páratlan számok szorzatát, és a szorzat gyökét, valamint a páros ciklusváltozók összegét és az átlagát! (*For1*)

```
#include <stdio.h>
#include <math.h>
int main()
{
  int i, db = 0;
  int osszeg = 0, szorzat = 1;
  double atlag, gyok;
```

```
    for( i = 0; i< 7; i++)
    {
      if ( i %2 == 0) { osszeg += i; db++; }
      else szorzat *= i;
    }
    atlag = (double)osszeg/db;
    gyok = sqrt(szorzat);
    printf("Paros szamok osszege     : %d\n",osszeg);
    printf("Paros szamok atlaga      : %lf\n",atlag);
    printf("Paratlan szamok szorzata: %d\n",szorzat);
    printf("Paratlan szamok szorzatanak gyoke: %lf\n",gyok);
    getchar();
    return 0;
}
```

A program futásának eredménye:

```
Paros szamok osszege     : 12
Paros szamok atlaga      : 3.000000
Paratlan szamok szorzata: 15
Paratlan szamok szorzatanak gyoke: 3.872983
```

Írjunk programot, amely végtelen **for** ciklusban addig olvas egész számokat, míg egymás után 2 db zérust nem olvasott! A második 0 adatot ne számoljuk bele az adatok darabszámába! Számláljuk meg az adatok darabszámát, és számítsuk ki a beolvasott pozitív adatok szorzatát! (*For2*)

Megjegyzés:
A végtelen **for** ciklus, amelyből hiányzik a kezdő, a végérték és a lépésköz:

```
for( ; ; )
{  }
```

```
#include <stdio.h>
int main()
{ int i, osszes_db = 0, adat, szorzat = 1, nulla = 0;
  for(;;)
  {
    osszes_db++;
    printf("Adat: "); scanf("%d",&adat);
    if (adat < 0) continue;
    if (adat > 0) szorzat *= adat;
    if (adat == 0) nulla++;
    if (nulla == 2) {  osszes_db--; break; }
    if (adat != 0) nulla = 0;
  }
  printf("Osszes adatok szama     : %d\n",osszes_db);
  printf("Pozitiv adatok szorzata: %d\n",szorzat);
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
Adat: 3
Adat: -4
Adat: 2
Adat: 0
Adat: 6
Adat: 0
Adat: 0
Osszes adatok szama     : 6
Pozitiv adatok szorzata: 36
```

4.4.3 A do-while ciklus

A **do-while** ciklus törzse egyszer, a feltétel tesztelése nélkül, mindig végrehajtódik. A ciklus addig működik, amíg a *kifejezés* igaz (nem 0). Az utasítás általános alakja:

```
do
{
  /* a ciklus törzse */
}while (kifejezés);
```

A C nyelvben ez az ún. hátul tesztelő ciklus.

Írjunk programot, amely a valós alapot pozitív illetve negatív egész kitevőre emeli! (*Dowhile_01*)

Az összefüggés, melynek alapján az alapot negatív hatványkitevőre emeljük:

$$a^{-n} = \frac{1}{a^n}$$

- Először kiszámítjuk az alapnak a pozitív hatványát úgy, hogy a **do-while** ciklus feltételben a kitevő abszolút értékét vizsgáljuk. Az abszolút érték meghatározása az ***abs()*** matematikai függvényt használjuk, melynek prototípusa a *MATH.H* fejállományban található.
- Ezután megvizsgáljuk a kitevő előjelét, amennyiben negatív, úgy a hatványértéknek a reciprokát képezzük.

```
#include <stdio.h>
#include <math.h>
int main()
{
  double alap, hatvany = 1;
  int kitevo, i;
  printf("Valos szamot emeljunk egesz kitevore \n");
  printf("\nAlap   : "); scanf("%lf",&alap);
  printf("Kitevo: "); scanf("%d",&kitevo);
  if (kitevo == 0)
    printf("\nHatvany erteke: %lf\n",hatvany);
```

```
  else
  {
    i = 1;
    hatvany = alap;
    do
    {
      hatvany *= alap;
      i++;
    }while (i < abs(kitevo));
    if (kitevo < 0)
      hatvany = 1./hatvany;
    printf("\nHatvany erteke: %lf\n",hatvany);
  }
  fflush(stdin);getchar();
  return 0;
}
```

A program futásainak eredménye:

```
Valos szamot emeljunk egesz kitevore

Alap   : 2
Kitevo: 3

Hatvany erteke: 8.000000


Valos szamot emeljunk egesz kitevore

Alap   : 2
Kitevo: -3

Hatvany erteke: 0.125000


Valos szamot emeljunk egesz kitevore

Alap   : 2
Kitevo: 0

Hatvany erteke: 1.000000
```

Írjunk programot, amely egy pozitív, 2-nél nagyobb számnak megkeresi az osztóit, illetve kijelzi, ha a szám prímszám! (*Dowhile2*)

Megjegyzés: A **do-while** ciklussal biztosítjuk, hogy megfelelő adatot olvassunk be!

```
#include <stdio.h>
int main()
{
  int szam, oszto, fele, db = 0;
  do
  {
    fflush(stdin);
    printf("Pozitiv egesz szam (>2): "); scanf("%d",&szam);
  }while (szam <=2);
```

```
  if (szam > 2)
  {
    fele = szam /2;
    printf("Osztoi\n");
    for (oszto = 2; oszto <= fele; oszto++)
    {
      if (szam % oszto == 0)
        {
          printf("   :%4d\n",oszto);
          db++;
        }
    }
    if (db == 0) printf("   :Primszam\n");
  }
  else printf("Hibas adat!\n");
  fflush(stdin);getchar();
  return 0;
}
```

A program futásainak eredménye:

```
Pozitiv egesz szam (>2): 21
Osztoi
  :   3
  :   7

Pozitiv egesz szam (>2): 13
Osztoi
  :Primszam
```

5. Származtatott típusok

Különböző feladatokat mutatunk be az egy- és többdimenziós tömbök, valamint a struktúrák kezelésére. A statikus és dinamikus adatstruktúrák használatát a megoldások alapján hasonlítjuk össze.

5.1 Tömbök, sztringek és mutatók

A strukturált (összetett) típusú változók egynél több értéket is tárolhatnak. A tömb adott számú, azonos típusú elemet tartalmazó adattípus. A tömb nélkülözhetetlen a programozás során, hiszen nagyban megkönnyíti több azonos adat kezelését. A tömb elemeinek típusa a **void** kivételével tetszőleges típus lehet.

5.1.1 Egydimenziós tömbök

Az egyetlen kiterjesztéssel rendelkező tömböket szokás vektoroknak nevezni. A többdimenziós tömbök esetén az elemek tárolása sorfolytonosan történik.
Egydimenziós tömb definíciója:

típus tömbnév[méret];

```
int main()
{
  int    x[10];   /* minden elem egész szám                */
  long   h[4];    /* minden eleme hosszú egész szám        */
  float  y[20];   /* minden eleme valós szám               */
  double w[5];    /* minden eleme duplapontosságú szám     */
  . . .
}
```

A C nyelvben a tömb indexe 0-val kezdődik, ezért **hibás** a deklarációban beírt értékre, mint indexre hivatkozni.

```
x[10] = 5;
```

Az *x* tömbnek valóban 10 darab eleme van, de az index csak 0-tól 9-ig változhat.

A C nyelv lehetővé teszi, hogy a tömböket a definiálásuk során konstans értékekkel inicializáljuk. Ez az értékadás eltér az egyszerű változók esetén használt megoldástól:

típus tömbnév[méret] = { *vesszővel tagolt konstansok*};

Néhány példa az inicializálásra:

```
int x[10] = { 2, 8, -1, 0, 6, 12, 9, 13, 4, 7 };
char gyumolcs[7] = { 'd', 'i', 'n', 'n', 'y', 'e' };
double y[4] = { 12.56, 42.6, -123.2, 0.05};
```

Írjunk programot, amely beolvassa a tömb aktuális elemszámát! Végezzünk ellenőrzést, ne fogadjunk el rossz elemszámot! Keressük meg a tömb legnagyobb és a legkisebb adatát! (*Tomb1*)

```
#include <stdio.h>
const int MERET = 60;
int main()
{ int i, ydb;
  double y[MERET], max, min;
  do
  {
    fflush(stdin);
    printf("Adatok szama (max. 60): "); scanf("%d",&ydb);
  }while (ydb <=0 || ydb > 60);
  for (i = 0; i<ydb; i++)
  {
    printf("%2d.adat: ",i); scanf("%lf",&y[i]);
  }
  max = y[0]; min = y[0];
  for (i = 1; i<ydb; i++)
  {
    if (y[i] > max) max = y[i];
    if (y[i] < min) min = y[i];
  }
  printf("\nLegnagyobb adata: %lf\n",max);
  printf("Legkisebb adata : %lf\n",min);
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
Adatok szama (max. 60): 4
 0.adat: 34.56
 1.adat: 12.43
 2.adat: 89.21
 3.adat: 23.78

Legnagyobb adata: 89.210000
Legkisebb adata : 12.430000
```

Írjunk programot, amely *MERET* darabszámú egész típusú adatot olvas be! Csak az *index* tömbben megadott elemeket jelenítsük meg, és számítsuk ki a szorzatukat! (*Tomb2*)

Megjegyzés:

- Egy tömb elemszámát meghatározhatjuk a **sizeof** operátorral úgy, hogy vesszük a tömb teljes méretét (**sizeof**(*tombnev*)) és osztjuk a *tombnev[0]* elemének a méretével:

```
index_db = sizeof(index)/sizeof(index[0]);
```

```
#include <stdio.h>
int index[] = {2,5,6,7,9};
const int index_db = sizeof(index)/sizeof(index[0]);
const int MERET = 10;
int main()
{ int x[MERET], i, szorzat = 1;
  printf("Adatok szama: %d\n",MERET);
  for (i = 0; i<MERET; i++)
  {
    printf("%d. adat: ",i); scanf("%d",&x[i]);
  }
  printf("\nOsszeszorzando elemek\n");
  printf("Index  Adat\n");
  for (i = 0; i<index_db; i++)
  {
    szorzat *= x[index[i]];
    printf("%3d %6d\n",index[i],x[index[i]]);
  }
  printf("Szorzat: %d\n",szorzat);
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
Adatok szama: 10
0. adat: 1
1. adat: 5
2. adat: 2
3. adat: 4
4. adat: 3
5. adat: 6
6. adat: 8
7. adat: 1
8. adat: 9
9. adat: 4
Osszeszorzando elemek
Index  Adat
  2      2
  5      6
  6      8
  7      1
  9      4
Szorzat: 384
```

Írjunk programot, amely beolvas egy elemszámot a tömb dinamikus létrehozásához! Számítsuk ki a pozitív adatok összegét és a negatív adatok átlagát! Kiíratás előtt vizsgáljuk meg az eredményeket! (*DimTomb*)

A C nyelvben szoros kapcsolat van a mutatók és a tömbök között. A mutatók segítségével a tömbök dinamikusan is létrehozhatók, számukra a kívánt nagyságú területet foglalhatunk le a ***malloc()*** vagy ***calloc()*** függvények segítségével:

```
void *malloc(size_t méret);
void *calloc(size_t elemszám, size_t elemméret);
```

A ***calloc()*** függvény a lefoglalt területet 0-ás bájtokkal tölti fel, ami főleg a sztringek esetén hasznos. A függvények használatához az *STDLIB.H* deklarációs állományt kell a programba beszerkeszteni a *#include* előfordító direktívával.

A függvények a lefoglalt területre mutató általános pointert (**void***) adnak vissza, melyet a bal oldalnak megfelelő típusra kell átalakítani. Ha a visszatérési érték *NULL*, akkor a memóriafoglalás nem sikerült.

A dinamikusan lefoglalt memóriaterületet a ***free()*** függvénnyel szabadítjuk fel. A ***free()*** függvény argumentum a felszabadítandó területre mutató pointer.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{ int *px = NULL, xdb, i, osszeg = 0, atlag_db = 0;
  double atlag = 0;
  printf("Adatok szama: "); scanf("%d",&xdb);
  px = (int*)malloc(xdb * sizeof(int));
  if (px == NULL)
  {
    printf("Memoriafoglalas nem sikerult!");
    fflush(stdin); getchar(); return 1;
  }
  for (i = 0; i<xdb; i++)
  { printf("%2d. elem: ",i); scanf("%d",&px[i]); }
  for (i = 0; i<xdb; i++)
  {
    if (px[i] > 0) osszeg += px[i];
    if (px[i] < 0) { atlag += px[i]; atlag_db++; }
  }
  if (osszeg > 0) printf("\nPozitiv adatok osszege: %d\n",osszeg);
  if (atlag_db > 0)
  {
    atlag = (double)atlag/atlag_db;
    printf("Negativ adatok atlaga: %lf\n",atlag);
  }
    free(px); px = NULL; /* már nem mutat sehová */
  fflush(stdin); getchar();
  return 0;
}
```

A program futásának eredménye:

```
Adatok szama: 6
 0. elem: 2
 1. elem: -4
 2. elem: 5
 3. elem: -2
 4. elem: 0
 5. elem: 3

Pozitiv adatok osszege: 10
Negativ adatok atlaga: -3.000000
```

5.1.2 Többdimenziós tömbök

A C nyelvben is használhatunk többdimenziós tömböket, melyek általános definíciója:

típus tömbnév[*méret1*][*méret2*]...[*méretn*];

Nézzük meg egy két dimenziós tömbök általános definícióját:

típus tömbnév [*méret1*][*méret2*];

ahol az első dimenzió (*méret1*) a tömb sorainak, míg a második (*méret2*) a tömb oszlopainak számát határozza meg. A kétdimenziós tömböt használják leggyakrabban.

```
int matrix[2][2] = {{ 2, 4},
                    { 3, 5}};
```

Az értékadás sorfolytonosan történik. A legegyszerűbb módszer, amikor egyszerűen soronként vesszővel elválasztva soroljuk fel az elemeket és {} zárójel világosan jelzi az értékadás. A második sor a tömb következő sorának felel meg. A kétdimenziós tömb első eleme *matrix[0][0]*, azaz 0. sorral kezdődik.

Az értékadás megfelel az alábbiaknak:

```
martrix[0][0] = 2; matrix[0][1] = 4;
martrix[1][0] = 3; matrix[1][1] = 5;
```

Néhány tömb definíció:

```
#define  M  10
int main()
{
  int     f[2][4];        /* minden elem egész szám       */
  double  d[M][M];        /* minden eleme valós szám  */
  . . .
}
```

Egy egész típusú 4×4-es mátrixot töltsünk fel úgy, hogy a felső háromszög 1-es, a főátló 2-es és az alsó háromszög 3-as értékeket vegyen fel. Számítsuk ki a főátlóbeli elemek szorzatát és a főátló alatti elemek összegét! (*Tomb2Dim*)

Megjegyzés:

- Egy szimmetrikus mátrix főátlóbeli elemeinek sor- és oszlopindexe megegyezik:
  `i == j`
- A mátrix sorait az *i*, az oszlopait pedig a *j* ciklusváltozóval érjük el a programban:
  `x[i][j]`

```
#include <stdio.h>
#define MERET 4
int main()
{ int x[MERET][MERET];
  int i, j, osszeg = 0, szorzat = 1;
  for (i = 0; i < MERET; i++)
   for (j = 0; j < MERET; j++)
   {
      if (i < j ) x[i][j] = 1;
```

```
        else if (i == j) x[i][j] = 2;
        else x[i][j] = 3;
    }
  printf("x tomb tartalma\n");
  for (i = 0; i < MERET; i++)
  {
   for (j = 0; j < MERET; j++)
     printf("%6d",x[i][j]);
   printf("\n");
  }
  for (i = 0; i < MERET; i++)
   for (j = 0; j < MERET; j++)
   {
     if ( i > j) osszeg += x[i][j];
     if ( i == j) szorzat *= x[i][j];
   }
 printf("\nSzorzat: %d\n",szorzat);
 printf("Osszeg : %d\n",osszeg);
 fflush(stdin); getchar();
 return 0;
}
```

A program futásának eredménye:

```
x tomb tartalma                    Szorzat: 16
     2     1     1     1           Osszeg : 18
     3     2     1     1
     3     3     2     1
     3     3     3     2
```

5.1.3 Karaktersorozatok (sztringek)

A C programban a szövegek tárolására statikus vagy dinamikus helyfoglalású karaktertömböket használunk.

A karaktertömbök a definícióban megadott értéknél egy karakterrel kevesebbet tudunk tárolni, mert a sztringet a tömbben az *EOS* (**E**nd **O**f **S**tring) jel a 0-ás bájt ('\0') zárja.

Karaktertömbök inicializálása sokkal biztonságosabban elvégezhető a sztring literálok (sztring konstansok) felhasználásával:

```
char s[10] = "Elefant";
char s[] = "Elefant";
```

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]
E	l	e	f	a	n	t	\0

Egy 20 elemű *neve* karaktertömbbe olvassuk be a tanuló nevét. Mivel a karaktertömb *neve* a nulladik elemre mutató pointer, ezért nem kell a ***scanf()*** függvénynél a címe jelet kitenni:

```
char nev[20],;
scanf("%s",neve);
```

A beolvasás után a *neve* karaktertömb tartalmát át kell másolni a *tan_neve* karaktertömb aktuális elemébe. A C nyelv nem teszi lehetővé a sztringek közvetlen értékadását, az ***strcpy()*** (*string copy* = sztring másolás) függvény segítségével másolhatjuk át a *neve* tartalmát a *tan_nev* karaktertömbbe, ehhez a *STRING.H* fejállomány beszerkesztése szükséges.

```
strcpy(tan_neve,neve);
```

A ***scanf()*** függvénnyel csak olyan karaktersorozatot olvashatunk be, amely nem tartalmaz szóközt.

Két sztring összehasonlítása az ***strcmp()*** függvénnyel lehetséges, amely zérus értékkel tér vissza, akkor a két sztring azonos:

```
if (strcmp(nev1, nev2) == 0) printf("azonos\n");
else printf("kulonbozo\n");
```

Írjunk programot, amelyben a ***gets()*** függvénnyel beolvasunk egy legfeljebb 79 karakter hosszú mondatot! A szavak végződése: .(szóköz, pont) lehet! Számláljuk meg a mondat szavainak számát! A karakterre mutató pointer növelésével írassuk ki a szavakat! (*CharTomb*)

Megjegyzés:

- A ***gets()*** függvény tetszőleges karaktersorozatot beolvas az <*Enter*> leütéséig. Az adatbevitel során figyelnünk kell arra, hogy ne olvassunk üres sztringet.

  ```
  while(gets(mondat)[0] == '\0');
  ```

- Hasznos még az input buffer törlése a ***gets()*** függvény olvasása előtt, hogy ne olvasson hibás adatot:

  ```
  fflush(stdin);
  ```

- A **char** tömb azonosítója a címet tartalmazza, ezért a karakterre mutató pointert a következő utasítással állítjuk rá:

  ```
  char *pmondat = mondat;
  ```

- A **for** ciklust a feltétel hamis értéke állítja le. Ez akkor következik be, amikor a feltételében szereplő *mondat* karaktertömb *i*-edik eleme a szöveg végét jelző nulla értékű ('\0') karaktert tartalmazza:

  ```
  for (i = 0; mondat[i]; i++)
  ```

  Nem szükséges a `mondat[i]!=0` feltétel használata.

- A mutató léptetése a cím növelését jelenti, azaz a következő karakterre lépünk:

  ```
  pmondat++;   vagy   ++pmondat;
  ```

```
#include <stdio.h>
int main()
{
  char mondat[80], *pmondat;
  int szavakszama = 0, i;
  printf("Mondat: ");
  fflush(stdin); /* input buffer törlése */
  while(gets(mondat)[0] == '\0');
  for (i = 0; mondat[i]; i++)
  {
     switch (mondat[i])
     {
       case ' ': szavakszama++; break;
       case '.': szavakszama++; break;
     }
  }
  printf("\nMondat szavainak szama: %d\n",szavakszama);
  printf("\nA mondat bontva: \n");
  pmondat = mondat;
  while (*pmondat)
  {
    switch (*pmondat)
    {
      case ' ': printf("\n"); break;
      case '.': printf("\n"); break;
      case '!': printf("\n"); break;
      case '?': printf("\n"); break;
      default: printf("%c",*pmondat);
   }
   pmondat++;
  }
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
Mondat: Ma jo lesz az ido.
Mondat szavainak szama: 5
A mondat bontva:
Ma
jo
lesz
az
ido
```

5.2 Felhasználó által definiált adattípusok

A C nyelvben is van lehetőség arra, hogy a nyelv meglévő típusait felhasználva újabb típusokat hozzunk létre.

5.2.1 A struct struktúratípus

A C nyelven a struktúra (**struct**) típus több, tetszőleges típusú (kivéve a **void** és a függvénytípust) adatok együttese. Általános formája:

struct *típusnév* {
típus1 tag1; /* *kezdőérték nélkül* */
típus2 tag2;
. . .
típusN tagN;
};

Írjunk programot, amely statikus és dinamikus helyfoglalású változókba beolvassa a *henger* típusú struktúra *r* és *h* adattagjait! Számítsuk ki a henger felszínét és térfogatát! (*Struct1*)

Megjegyzés:

- A **struct** és a típusnévvel is definiálhatunk: `struct h h1;`
- A struktúra befejező } zárójele mögött is adhatunk típust.
  `henger h2;`
- Statikus struktúra esetében az adattagokat ponttal érjük el:
  `h2.r = 2;`
- Dinamikus struktúra esetén, először helyet kell foglalni a struktúra számára a memóriában a ***malloc()*** függvénnyel. A tagokat a nyíl -> (kötőjel és nagyobb jel) operátorral érjük el. A dinamikus helyfoglalású struktúrát a ***free()*** függvénnyel szabadítjuk fel:

```
struct h *ph3 = (struct h*)malloc(sizeof(struct h));
ph3->r = 4; ph3->h = 5;
free(ph3);
```

A típusnév későbbi használat esetén a típusdefiníció {} közötti törzsét teljes értékűen helyettesíti. (A típusnév használata hasonló az **enum** típusnál is.)

Használjuk az alábbi definíciót, amely a henger adatait tartalmazza:

```
typedef struct h{
                 double r, h;
                 double terf, felsz;
                }henger;
```

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define PI 3.141592654
typedef struct h{
                 double r, h;
                 double terf, felsz;
                }henger;
```

```
int main()
{ struct h h1;
  henger h2, *ph3;
  h1.r = 1; h1.h = 1;
  h1.terf = pow(h1.r,2) * PI * h1.h;
  h1.felsz = 2 * pow(h1.r,2) * PI + 2 * h1.r * PI * h1.h;
  printf("1. Henger\n");
  printf("sugara: %8.2lf  magassaga: %8.2lf\n",h1.r,h1.h);
  printf("felszine : %8.2lf\n",h1.felsz);
  printf("terfogata: %8.2lf\n",h1.terf);
  printf("\n2. Henger\n");
  printf("sugara: "); scanf("%lf",&h2.r);
  printf("magassaga: "); scanf("%lf",&h2.h);
  h2.terf = pow(h2.r,2) * PI * h2.h;
  h2.felsz = 2 * pow(h2.r,2) * PI + 2 * h2.r * PI * h2.h;
  printf("felszine : %8.2lf\n",h2.felsz);
  printf("terfogata: %8.2lf\n",h2.terf);
  ph3 = (henger*)malloc(sizeof(henger));
  if (ph3 == NULL)
  { printf("Memoriafoglalas nem sikerult!\n"); getchar(); return 1; }
  ph3->r = 4; ph3->h = 5;
  ph3->terf = pow(ph3->r,2) * PI * ph3->h;
  ph3->felsz = 2 * pow(ph3->r,2) * PI + 2 * ph3->r * PI * ph3->h;
  printf("\n3. Henger\n");
  printf("sugara: %8.2lf  magassaga: %8.2lf\n",ph3->r,ph3->h);
  printf("felszine : %8.2lf\n",ph3->felsz);
  printf("terfogata: %8.2lf\n",ph3->terf);
  free(ph3);
  fflush(stdin); getchar();
  return 0;
}
```

A program futásának eredménye:

```
1. Henger
sugara:     1.00  magassaga:     1.00
felszine :    12.57
terfogata:     3.14

2. Henger
sugara: 2
magassaga: 3
felszine :    62.83
terfogata:    37.70

3. Henger
sugara:     4.00  magassaga:     5.00
felszine :   226.19
terfogata:   251.33
```

5.2.2 Struktúratömbök

A struktúratömbök definiálása, teljesen megegyezik más típusú tömbök definíciójával, amely – mint tudjuk – lehet statikus, vagy dinamikus. Ebben az esetben azonban a tömb elemei struktúrák.

Írjunk programot, amely *N* elemű, *henger* típusú statikus *h1* struktúratömböt konstansokkal tölt fel, majd beolvassa a dinamikusan létrehozott *ph2* henger *r* és *h* adattagjait! Számítsuk ki a hengerek felszínét és térfogatát! Írassuk ki táblázatosan az eredményeket! (*StructTomb*)

```
typedef struct h{
                  double r, h;
                  double terf, felsz;
                 }henger;
const int N = 3;
/* kezdőértékadás a tömbnek, illetve a struktúráknak */
henger h1[] ={ {1, 1},
               {2, 3},
               {1.5, 2.5}
             };
```

Megjegyzés:

- Mivel példában szereplő *h1* tömb elemei *henger* típusú struktúrák, a struktúra tagjait a következő módon érhetjük el: `h1[i].r`, azaz a tömbből az indexszel kiválasztjuk a struktúrát, majd pedig a pontoperátorral kijelöljük a struktúra adattagját, függetlenül attól, hogy dinamikusan hoztuk-e létre a tömböt.
- A *ph2* a *henger* struktúrára mutató pointer. Ha *N* darab struktúrának foglalunk helyet, akkor a ***malloc()*** függvényt kell hívnunk:

  ```
  henger *ph2;
  ph2 = (henger*)malloc(N*sizeof(henger));
  ```

  Az adattagok elérése szintén a pontoperátorral történik: `ph2[i].r`
- A dinamikus helyfoglalású tömböt a ***free()*** függvénnyel töröljük a memóriából, argumentumként megadva a felszabadítandó terület mutatóját, majd a NULL értékadással jelezzük, hogy az már sehova sem mutat.

  ```
  free(ph2); ph2 = NULL;
  ```

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define PI 3.141592654
typedef struct h{
                  double r, h;
                  double terf, felsz;
                 }henger;
const int N = 3; /* a struktúratömb mérete */
/* konstansokkal feltöltött struktúratömb */
henger h1[] ={ {1, 1},
               {2, 3},
               {1.5, 2.5}
             };
int main()
{ int i;
  henger *ph2 = NULL;
  printf("Statikus henger adatai\n");
```

```
  printf("Adatok szama: %d\n",N);
  for (i = 0; i<N; i++)
  {
    printf("\nsugar: %8.2lf\n",h1[i].r); /* megjeleníti az adatokat */
    printf("magassag: %8.2lf\n",h1[i].h);
    h1[i].terf = pow(h1[i].r,2) * PI * h1[i].h;/* térfogat számítása */
    h1[i].felsz = 2 * pow(h1[i].r,2) * PI +
                  2 * h1[i].r * PI * h1[i].h; /* felszín számítása */
  }
  printf("\nTerfogat    Felszin\n");printf("--------------------\n");
  for (i = 0; i<N; i++)   /* az eredmények megjelenítése */
  {
    printf("%8.2lf   %8.2lf\n",h1[i].terf,h1[i].felsz);
  }
  ph2 = (henger*)malloc(N*sizeof(henger)); /* memóriafoglalás */
  if (ph2 == NULL)
  {
    printf("Memoriafoglalas nem sikerult!\n");
    fflush(stdin); getchar(); return 1;
  }
  printf("\nDinamikus henger adatai\n");
  printf("Adatok szama: %d\n",N);
  for (i = 0; i<N; i++)
  {
    printf("\nsugar: "); scanf("%lf",&ph2[i].r);
    printf("magassag: "); scanf("%lf",&ph2[i].h);
    ph2[i].terf = pow(ph2[i].r,2) * PI * ph2[i].h;
    ph2[i].felsz = 2 * pow(ph2[i].r,2) * PI +
                   2 * ph2[i].r * PI * ph2[i].h;
  }
  printf("\nTerfogat    Felszin\n");
  printf("--------------------\n");
  for (i = 0; i<N; i++)
  {  printf("%8.2lf   %8.2lf\n",ph2[i].terf,ph2[i].felsz); }
  free(ph2); fflush(stdin); getchar();
  return 0;
}
```

A program futásának eredménye:

```
Statikus henger adatai
Adatok szama: 3

sugar:      1.00
magassag:      1.00

sugar:      2.00
magassag:      3.00

sugar:      1.50
magassag:      2.50

Terfogat    Felszin
--------------------
    3.14       12.57
   37.70       62.83
   17.67       37.70
```

```
Dinamikus henger adatai
Adatok szama: 3

sugar: 1
magassag: 1

sugar: 3.2
magassag: 1.7

sugar: 4.2
magassag: 5.6

Terfogat    Felszin
----------------------
    3.14       12.57
   54.69       98.52
  310.34      258.62
```

Írjunk programot, amely értéket ad a *henger_ft* típusú *heng1* változó *r* és *h* tagjainak, melyekett a *h1* adattagon keresztül ér el! A *h1* tag szintén egy struktúra. Számítsuk ki a henger felszínét és térfogatát! (*Struct2*)

```
typedef struct ha{
                  double r, h;
                }henger_adatok;
typedef struct hf{
                   henger_adatok h1;
                   double terf, felsz;
                }henger_ft;
```

Megjegyzés:

- Ha egy struktúra adattagja szintén egy struktúra, akkor az adatok elérése a következő módon történik:

```
henger_ft heng1;
heng1.h1.r = 1; heng1.h1.h = 1;
```

Azaz első pont után a tagnév, és a második pont után a beágyazott struktúra tagneve következik.

```
#include <stdio.h>
#include <math.h>
#define PI 3.141592654
typedef struct ha{ double r, h;
                }henger_adatok;
typedef struct hf{  henger_adatok h1;
                   double terf, felsz;
                }henger_ft;
int main()
{ henger_ft heng1;
  heng1.h1.r = 1; heng1.h1.h = 1;
  heng1.terf = pow(heng1.h1.r,2) * PI * heng1.h1.h;

  heng1.felsz = 2 * pow(heng1.h1.r,2) * PI +
                2 * heng1.h1.r * PI * heng1.h1.h;
  printf("A henger felszinenek es terfogatanak szamitasa\n");
  printf("sugar    : %8.2lf",heng1.h1.r);
  printf("\nmagassag: %8.2lf\n",heng1.h1.h);
  printf("\nfelszin : %8.2lf\n",heng1.felsz);
  printf("terfogat: %8.2lf",heng1.terf);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
A henger felszinenek es terfogatanak szamitasa
sugar    :     1.00
magassag:      1.00

felszin :     12.57
terfogat:      3.14
```

6. Függvények

A C program nélkülözhetetlen függvénye a ***main()*** függvény, amely mellett általában további függvények oldják meg a feladatot, és a ***main()*** feladata ezeknek a függvényeknek az aktiválása.

6.1. A függvények definiálása

A függvény definíció általános formája:

```
visszatérési típus függvénynév(paraméterdeklarációs lista)
{
    /* a függvény törzse*/
    lokális definíciók és deklarációk
    utasítások
}
```

Ha a függvény definíciója a ***main()*** függvény előtt helyezkedik el és megelőzi a hívás helyét, akkor ez egyben a függvény deklarációja is.

Az összes példában a saját függvényeket a ***main()*** függvény után helyezzük, így a függvény deklarációját mindig meg kell adnunk a ***main()*** függvény előtt.

A deklaráció a függvény nevét, paramétereinek és visszatérési értékének típusát tartalmazza, mely után pontosvesszőt kell tenni. A függvények oly módon megadott deklarációját prototípusnak nevezzük.

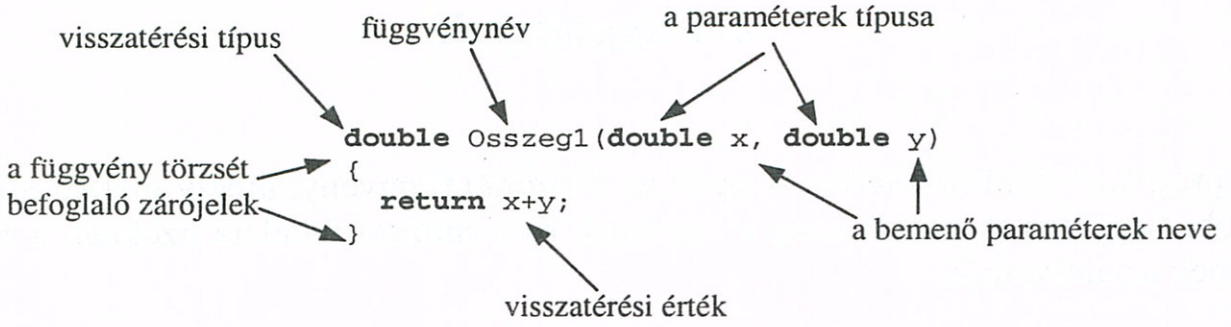
```
visszatérési típus függvénynév(paraméterdeklarációs lista);
```

A **return** utasításnak két fontos szerepe van:

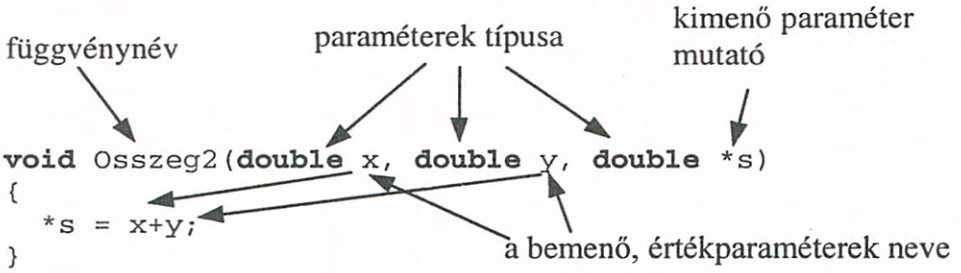
- hatására a hívott függvény visszatér a hívás helyére,
- az utasításban szereplő kifejezés, mint függvényérték (visszatérési érték) jelenik meg.

A C nyelv függvényeit az alábbiak szerint is osztályozhatjuk:

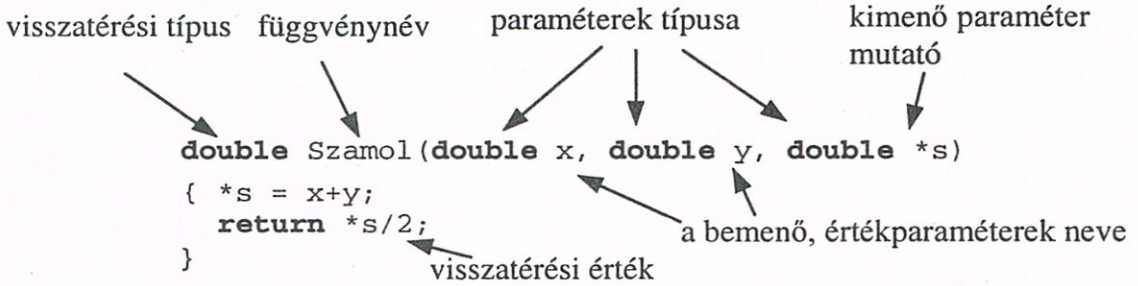
1. A függvény paraméterlistáján szereplő változók értékeivel számol, és egyetlen eredményt ad vissza a **return** segítségével. Így ez a függvény aktiválható egy értékadó utasítás jobb oldalán, vagy ***printf()***-ben, ahol a visszaadott érték azonnal kiíratható.



2. A függvénynek nincs visszatérési értéke (**void**), a paraméterlistáján szereplő pointer segítségével a függvény vissza tud adni értéket, természetesen egynél többet is. Ezeknek a függvényeknek hívása hasonlít a más nyelvekben megvalósított eljárásokéhoz, mivel a függvényhívás önálló (kifejezés) utasítás.



3. A függvénynek van visszatérési értéke, de a mutató-paraméterein keresztül is ad vissza értékeket.



4. A függvénynek nincsenek paraméterei és nincsen visszatérési értéke, de lehet benne kifejezés nélküli **return**.

6.2 Különböző típusú paraméterek

Ebben a fejezetben olyan feladatokat mutatunk be, amelyekben a függvények a legkülönfélébb típusú paramétereket kezelnek.

6.2.1 Aritmetikai típusú paraméterek

Írjunk főfüggvényt a C nyelv függvény fajtáit bemutató függvényekhez, a ***main()*** függvényből a függvényeket hívjuk argumentumaikkal, az eredményeket formázatlanul és formázottan egyaránt jelenítsük meg! (*MintaFg*)

```
/* a program fejállományai */
#include <stdio.h>
#include <stdlib.h>
```

A mintaprogramban szereplő függvények prototípusi és feladata

Írjunk *Osszeg1()* függvényt, amely az argumentumlistán megadott két valós számot összeadja, és a **return** mellett, függvényértékként adja vissza!

```
double Osszeg1(double x, double y);
```

Készítsünk *Osszeg2()* függvényt, amely az argumentumlistán megadott két valós számot összeadja, és az *s* mutató segítségével adja vissza!

```
void Osszeg2(double x, double y, double *s);
```

Írjunk *Szamol()* függvényt, amely az argumentumlistán megadott két valós számot összeadja, és az *s* mutató segítségével adja vissza, valamint képezi a két valós szám átlagát, melyet a **return** mellett, függvényértékként ad vissza!

```
double Szamol(double x, double y, double *s);
```

Készítsünk *Fejlec1()* függvényt, amely megjeleníti a feladat fejlécét, és tartalmaz egy kifejezés nélküli **return** utasítást!

```
void Fejlec1(void);
```

Írjunk *Fejlec1()* függvényt, amely az adatok beolvasására jelenít meg fejlécet:

```
void Fejlec2(void);
```

A feladat megoldása:

```
/*MINTAFG*/
int main()
{ double a = 1, b = 2, ered1, ered2, ered3, *ered4, atlag1, atlag2;
  ered1 = Osszeg1(a,b);
  Fejlec1(); /* Fejlec() függvény aktiválása */
  printf("Osszeg1 eredmenye: %lf\n",ered1);
  printf("Osszeg1 printf()-ben hivasa: %lf\n",Osszeg1(2,3));

  Fejlec2();
  printf("1.adat: "); scanf("%lf",&a); /* adatok beolvasása */
  printf("2.adat: "); scanf("%lf",&b);
  Osszeg2(a,b,&ered2);  /* visszatérési érték nélküli aktiválás */
  printf("Osszeg2 eredmenye: %lf\n",ered2);

  atlag1 = Szamol(a,b,&ered3); /*ered3 címe, mint kimenő paraméter */
  printf("\nSzamol 1. hivas eredmenye\n");
  printf("atlag1: %9.2e\n",atlag1); /* eredmények megjelenítése */
  printf("ered3 : %9.2e\n",ered3);

  printf("\nAz ered4 mutatonak memoriaterulet foglalasa.\n");
  ered4 = (double *)malloc(sizeof(double));
  if (ered4 == NULL)
  { printf("Memoriafoglalas nem sikerult!\n");
    fflush(stdin); getchar(); return 1;
  }
  /* ered4 mutato(cím), kimenő paraméter */
  atlag2 = Szamol(a,b,ered4);
```

```
  printf("\nSzamol 2. hivas eredmenye\n");
  printf("atlag2: %8.2lf\n",atlag2); /* eredmények megjelenítése */
  printf("*ered4: %8.2lf\n",*ered4);
  printf("\nAz ered4 mutato memoriateruletenek felszabaditasa.\n");
  free(ered4); /* memóriaterület felszabadítása */
  fflush(stdin); getchar();
  return 0;
}
```

A program futásának eredménye:

```
C fuggveny fajtainak bemutatasa

Osszeg1 eredmenye: 3.000000
Osszeg1 printf()-ben hivasa: 5.000000

Adatok beolvasasa
1.adat: 4
2.adat: 6
Osszeg2 eredmenye: 10.000000

Szamol 1. hivas eredmenye
atlag1: 5.00e+000
ered3 : 1.00e+001

Az ered4 mutatonak memoriaterulet foglalasa.

Szamol 2. hivas eredmenye
atlag2:     5.00
*ered4:    10.00

Az ered4 mutato memoriateruletenek felszabaditasa.
```

A függvények megvalósításának magyarázata:

Az *Osszeg1()* függvény az *x* és *y* paraméter összegét a **return** mellett adja vissza függvényértékként.

```
double Osszeg1(double x, double y)
{
  return x+y;
}
```

Hívása: `ered1 = Osszeg1(a,b);`

Amelyik függvénynek van visszatérési értéke, az eredmény tárolása helyett közvetlenül kiírathatjuk az eredmény értékét, ha meghívhatjuk a ***printf()***-ben.

```
printf("Osszeg1 printf()-ben hivasa: %lf\n",Osszeg1(2,3));
```

Az *Osszeg2()* az *x* és *y* paraméter összegét az *s* mutatóval adja vissza a paraméterlistán.

```
void Osszeg2(double x, double y, double *s)
{
  *s = x+y;
}
```

Hívása: `Osszeg2(a,b,&ered2);`

A *Szamol()* függvény az *x* és *y* paraméter összegét az *s* mutatóval adja vissza a paraméterlistán, a két paraméter átlagát pedig a **return** mellett adja vissza:

```
double Szamol(double x, double y, double *s)
{
  *s = x+y;
  return *s/2;
}
```

Az argumentumlistán a pointert a statikus adat címével helyettesítjük:

Hívása: `atlag1 = Szamol(a,b,&ered3);`

A paraméterlistán a pointert a pointer nevével (címével) helyettesítjük:
Hívása: `atlag2 = Szamol(a,b,ered4);`

A *Fejlec1()* függvény a program eredményeihez egy fejlécet jelenít meg, és tartalmaz egy kifejezés nélküli **return** utasítást, melynek most el is hagyható.

```
void Fejlec1(void)
{
  printf("C fuggveny fajtainak bemutatasa\n\n");
  return; /*ures */
}
```

Hívása: `Fejlec1();`

A *Fejlec2()* függvény fejlécet jelenít meg az adatok beolvasásához.

```
void Fejlec2(void)
{
  printf("\nAdatok beolvasasa\n");
}
```

Hívása: `Fejlec2();`

6.2.2 Felhasználói típusú paraméterek

Írjunk programot, amely függvénnyel olvassa be a henger sugarának és magaságának hosszát, és választhatóan kiszámítja a felszínét vagy a térfogatát! Az adatok tárolására használjunk struktúrát! (*Struct1*)

```
#include <stdio.h>
#include <math.h>
#define PI 3.141592654
typedef struct {
                 double sugar, magassag;
               }Henger;
Henger Olvas(void);
double Szamol(Henger x, int valaszt);
```

```
int main()
{
  Henger k;
  k = Olvas();
  printf("\nHenger felszine  : %lf\n",Szamol(k,1));
  printf("Henger terfogata : %lf\n",Szamol(k,2));
  fflush(stdin); getchar();
  return 0;
}
```

Az *Olvas()* függvény egy *Henger* típusú struktúra *sugar* és a *magassag* adattagjainak ad értéket, majd a struktúrát függvényértékként visszaadja.

```
Henger Olvas(void)
{
  Henger x;
  printf("Henger sugara    : "); scanf("%lf",&x.sugar);
  printf("Henger magassaga: "); scanf("%lf",&x.magassag);
  return x;
}
```

A *Szamol()* függvény egy *Henger* típusú struktúrát és egy egész számot (*valaszt*) fogad paraméterként. Ha a *valaszt* paraméter értéke 1, akkor a henger felszínét, ha 2, akkor a térfogatát számítja ki, melyet függvényértékként ad vissza.

```
double Szamol(Henger x, int valaszt)
{
  double ered;
  switch(valaszt)
  {
    case 1: ered = 2*pow(x.sugar,2)*PI+
                   2*x.sugar*PI*x.magassag; break;
    case 2: ered = pow(x.sugar,2)*PI*x.magassag; break;
  }
  return ered;
}
```

A program futásának eredménye:

```
Henger sugara    : 1
Henger magassaga: 1

Henger felszine  : 12.566371
Henger terfogata : 3.141593
```

Írjunk programot, amely kétféle függvénnyel olvassa be a *Kor* struktúra középpontjának *x*, *y* koordinátáit és sugarát (*r*), majd kiszámítja a kör területét a *ter* adattagba! Az első térnegyeden (x>0, y>0) adja össze a körök területét, a második térnegyeden (x<0, y>0) és a harmadik térnegyeden (x<0, y<0) számlálja meg az odaeső köröket! (*Struct2*)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define PI 3.141592654
typedef struct {
                  int x, y, r;
                  double ter;
                }Kor;

Kor *Feltolt1(int *db);
int Feltolt2(Kor **w);
int Szamlal(Kor *w, int db, double *ter, int *negt);

int main()
{
  int db1, db2, pdb1, pdb2, ndb1, ndb2;
  double ter1, ter2;
  Kor *k1 = NULL, *k2 = NULL;
  k1 = Feltolt1(&db1); /* első struktúratömb feltöltése */
  if (k1 == NULL)
  { printf("Memoriafoglalas nem sikerult!\n");
    fflush(stdin);getchar();return 1;
  }
  pdb1 = Szamlal(k1, db1, &ter1, &ndb1);
  printf("\nFeltolt1 hivasanak eredmenye\n");
  printf("1. ternegyedben a korok terulete: %lf\n",ter1);
  printf("2. ternegyedben a korok szama: %d\n",pdb1);
  printf("3. ternegyedben a korok szama: %d\n",ndb1);
  db2 = Feltolt2(&k2); /* második struktúratömb feltöltése */
  if (k2 == NULL)
  { printf("Memoriafoglalas nem sikerult!\n");
    fflush(stdin);getchar();return 1;
  }
  pdb2 = Szamlal(k2, db2, &ter2, &ndb2);
  printf("\nFeltolt2 hivasanak eredmenye\n");
  printf("1. ternegyedben a korok terulete: %lf\n",ter2);
  printf("2. ternegyedben a korok szama: %d\n",pdb2);
  printf("3. ternegyedben a korok szama: %d\n",ndb2);
  free(k1);
  free(k2);
  fflush(stdin); getchar();
  return 0;
}
```

A *Feltolt1()* függvény beolvassa a *Kor* struktúra darabszámát, melyet a paraméterlistán a *db* mutatóban ad vissza. Memóriahelyet foglal a *Kor* típusú *w* struktúra tömbnek, és feltölti adatokkal. A struktúratömb mutatóját, függvényértékként adja vissza.

```
Kor *Feltolt1(int *db)
{
  int i;
  Kor *w = NULL;
  printf("Korok szama: "); scanf("%d",db);
  w = (Kor*)malloc(*db * sizeof(Kor));
```

```
  if (w != NULL)
  { for (i = 0; i<*db; i++)
    { printf("\nx-koordinata: "); scanf("%d",&w[i].x);
      printf("y-koordinata: "); scanf("%d",&w[i].y);
      printf("sugar       : "); scanf("%d",&w[i].r);
      w[i].ter = pow(w[i].r,2)*PI;
    }
  }
  return w;
}
```

A *Feltolt2()* függvény beolvassa a *Kor* struktúra darabszámát, melyet függvényértékként ad vissza. Memóriahelyet foglal a *Kor* típusú *ws* struktúra tömbnek, feltölti adatokkal. A struktúratömb mutatóját a *w* paraméterben adja vissza. Mutatót csak dupla csillagos mutatóban lehet visszaadni.

```
int Feltolt2(Kor **w)
{
  int db, i;
  Kor *ws = NULL;
  printf("\nKorok szama: "); scanf("%d",&db);
  ws = (Kor*)malloc(db * sizeof(Kor));
  if (ws != NULL)
  { for (i = 0; i<db; i++)
    {
      printf("\nx-koordinata: "); scanf("%d",&ws[i].x);
      printf("y-koordinata: "); scanf("%d",&ws[i].y);
      printf("sugar       : "); scanf("%d",&ws[i].r);
      ws[i].ter = pow(ws[i].r,2)*PI;
    }
  }
  *w = ws;
  return db;
}
```

A *Szamlal()* függvény a paraméterként megkapja a Kor struktúrák tömbjének kezdőcímét (*w*) az elemek számát (*db*). Ennek ismeretében összeadja az első térnegyedbe eső körök területét (**ter*), megszámlálja a 3. térnegyedbe eső köröket (**negt*) és a két eredményt a paraméterlistán adja vissza. A második térnegyedbe eső köröket is megszámlálja (lokálisan deklarált *db2t*) és ezt függvényértékként adja vissza.

```
int Szamlal(Kor *w, int db, double *ter, int *negt)
{ int i, db2t = 0;
  *ter = 0;
  *negt = 0;
  for (i = 0; i<db; i++)
  {
    if (w[i].x > 0 && w[i].y > 0) *ter += w[i].ter;
    if (w[i].x < 0 && w[i].y > 0) db2t++;
    if (w[i].x < 0 && w[i].y < 0) (*negt)++;
  }
  return db2t;
}
```

A program futásainak eredménye:

```
Korok szama: 3

x-koordinata: 1
y-koordinata: 1
sugar       : 1

x-koordinata: -2
y-koordinata: 2
sugar       : 3

x-koordinata: -2
y-koordinata: -3
sugar       : 4

Feltolt1 hivasanak eredmenye
1. ternegyedben a korok terulete: 3.141593
2. ternegyedben a korok szama: 1
3. ternegyedben a korok szama: 1

Korok szama: 1

x-koordinata: 2
y-koordinata: 2
sugar       : 12

Feltolt2 hivasanak eredmenye
1. ternegyedben a korok terulete: 452.389342
2. ternegyedben a korok szama: 0
3. ternegyedben a korok szama: 0
```

6.2.3 Tömbök átadása függvénynek

Írjunk programot, amelyben a *VOlvas()* függvénnyel beolvassuk a valós típusú tömb adatait, és a *PozNegSzamol()* függvénnyel kiszámítjuk a pozitív adatok összegét és a negatív adatok átlagát! (*Tomb1*)

```
#include <stdio.h>
typedef double vekt[20];
int VOlvas(vekt x);
double PozNegSzamol(vekt x, int db, double *PozOssz);
int main()
{
  vekt v;
  int v_db;
  double POssz, NAtl;
  v_db = VOlvas(v);
  NAtl = PozNegSzamol(v,v_db,&POssz);
  if (POssz > 0)
    printf("\nPozitiv elemek osszege: %lf\n",POssz);
  if (NAtl < 0)
    printf("Negativ elemek atlaga : %lf\n",NAtl);
  fflush(stdin);getchar();
  return 0;
}
```

A *VOlvas()* függvény paraméterben adja vissza az adatokkal feltöltött valós típusú *x* tömböt, a tömb elemeinek darabszámát (*db*), pedig függvényértékként szolgáltatja. A függvény hibás darabszámot nem fogad el.

```
int VOlvas(vekt x)
{
  int i, db;
  do
  {
    fflush(stdin);
    printf("Adatok szama: "); scanf("%d",&db);
  }while (db <=1 || db>20);
  for (i = 0; i<db; i++)
  {
    printf("%2d. adat: ",i); scanf("%lf",&x[i]);
  }
  return db;
}
```

A *PzNegSzamol()* függvény a *x* tömb és a *db* darabszám ismeretében a negatív elemeket és a pozitív elemeket külön összeadja és megszámlálja. A pozitív elemek összegét a paraméterlistán adja vissza, ha van negatív összeg, akkor kiszámítja az átlagot, egyébként zérust ad vissza függvényértékként.

```
double PozNegSzamol(vekt x, int db, double *PozOssz)
{
  int i, pdb = 0, ndb = 0;
  double NegAtl = 0;
  *PozOssz = 0;
  for (i = 0; i<db; i++)
  {
    /* ha a tömb eleme pozitív, pozitiv összeg számítása */
    if (x[i] < 0) { NegAtl += x[i]; ndb++; }
       /*  ha a tömb eleme negatív, negatív összeg számítása
        és a darabszám számlálása */
    if (x[i] > 0) { *PozOssz += x[i]; pdb++; }
  }
  /* vizsgálat arra, hogy volt-e negatív eleme a tömbnek */
  if (ndb > 0) NegAtl /= ndb; /*negatív átlag számítása */
 return NegAtl;
}
```

A program futásának eredménye:

```
Adatok szama: 5
 0. adat: 1
 1. adat: 2
 2. adat: 3
 3. adat: -4
 4. adat: -5

Pozitiv elemek osszege: 6.000000
Negativ elemek atlaga : -4.500000
```

Írjunk programot, amelyben a *TOlvas()* függvénnyel beolvassuk a saját típussal definiált valós tömböt, és a *Szamol()* függvénnyel számítsuk ki a tömb elemeinek összegét, átlagát és megszámláljuk, hogy a tömbnek hány eleme esik az átlag fölé! (*Tomb2*)

```
#include <stdio.h>
typedef int* tomb;
tomb TOlvas(int *db);
double Szamol(tomb b, int db, int *nagyobb, int *osszeg);
int main()
{
  int  x_db, x_osszege, x_nagyobb;
  double x_atlaga;
  tomb x;
  x = TOlvas(&x_db);
  if (x == NULL)
  {
     printf("Memoriafoglalas nem sikerult!\n");
     fflush(stdin); getchar(); return 1;
  }
  x_atlaga = Szamol(x, x_db, &x_nagyobb, &x_osszege);
  printf("\nTomb elemeinek osszege: %d\n",x_osszege);
  printf("Tomb elemeinek atlaga : %8.2lf\n",x_atlaga);
  printf("Atlagnal nagyobb elemek szama: %d\n",x_nagyobb);
  free(x);
  fflush(stdin); getchar();
  return 0;
}
```

A *TOlvas()* függvény tömb aktuális elemeinek *db* darabszámát a függvény paraméterlistán, és az adatokkal feltöltött dinamikus *b* tömb címét pedig függvényértékként adja vissza.

```
tomb TOlvas(int *db)
{
  int i;
  tomb b = NULL;
  printf("Adatok szama: "); scanf("%d",db);
  b = (tomb)malloc(*db * sizeof(int));
  if (b != NULL)
  {
    for (i = 0; i<*db; i++)
    {
      printf("%2d. adat: ",i); scanf("%d",&b[i]);
    }
  }
  return b;
}
```

A *Szamol()* függvény a *db* elemszámú *tomb* típusú *b* tömbnek az ismeretében kiszámítja az elemek összegét és az átlagát. Megszámlálja azokat az elemeket, melyek az átlagnál nagyobbak. Az összeget és az átlagnál nagyobb elemek számát a paraméterlistán, az átlagot pedig függvényértékként adja vissza.

```
double Szamol(tomb b, int db, int *nagyobb, int *osszeg)
{
  int i;
  double atlag;
  *nagyobb = 0;
  *osszeg = 0;

  for (i = 0; i<db; i++)
  {
    *osszeg += b[i];
  }
  atlag = (double)*osszeg/db;
  for (i = 0; i<db; i++)
  {
    if (b[i] > atlag) (*nagyobb)++;
  }
  return atlag;
}
```

A program futásának eredménye:

```
Adatok szama: 5
 0. adat: 1
 1. adat: 2
 2. adat: 3
 3. adat: 4
 4. adat: 5

Tomb elemeinek osszege: 15
Tomb elemeinek atlaga :     3.00
Atlagnal nagyobb elemek szama: 2
```

Írjunk programot, amely függvények használatával kockadobást szimulál! A szimulációhoz használjon véletlen számokat, készítsen statisztikát az eredményekről és jelenítse meg azt! (*KockaDob*)

```
#include <stdio.h>
#include <stdlib.h>
typedef int dob[7];
void Kockadobas(dob kocka, int dobas);
void Megjelenit(dob kocka);
void Kezdoertek(void);
int main()
{ dob k;
  Kezdoertek();
  Kockadobas(k,1000);
  Megjelenit(k);
  getchar();
  return 0;
}
```

A *Kezdoertek()* függvény a ***srand()*** könyvtári függvénnyel a véletlenszám-generátornak közel véletlen kezdőértéket állít be.

```
void Kezdoertek()
{
  srand((unsigned) time(NULL));
}
```

A *Kockadobás()* függvény *dob* típusú *kocka* tömbbe *dobas* darab kockadobást szimulál.

```
void Kockadobas(dob kocka, int dobas)
{ int i, index;
  for (i = 1; i<7; i++)
    kocka[i] = 0;
  for (i = 1; i<=dobas; i++)
  {
    index = rand()%6 + 1;
    kocka[index]++;
  }
}
```

A *Megjelenit()* függvény a paraméterlistán megadott *dob* típusú *kocka* tömb elemeinek az indexét és tartalmát jeleníti meg.

```
void Megjelenit(dob kocka)
{ int i, ossz_db = 0;
  for (i = 1; i<=6; i++)
  {
    printf("%2d   %4d\n",i, kocka[i]);
    ossz_db += kocka[i];
  }
  printf("-----------\n");
  printf("  %7d\n",ossz_db);
}
```

A program futásának eredménye:

```
 1    161
 2    162
 3    150
 4    163
 5    177
 6    187
-----------
     1000
```

6.2.4 Sztringparaméterek

Írjunk programot, amely függvények hívásával egy adott karaktersorozat minden páros sorszámú karaktere mögé egy megadott karaktert beszúr, valamint a karaktertömböt kiírja a képernyőre! A feladat megoldása során az eredeti és a módosított karaktersorozatra is szükségünk van. (*KarBeszur*)

Megjegyzés: C nyelvben a karaktersorozatok tárolására karaktertömböt használunk, melyben a karaktereket egy nullás bájt zárja.

```
#include <stdio.h>
#include <string.h>
typedef char string[80];
char* KarBeszur(string szoveg, char ch);
void SzovegKiir(char *ki);
int main()
{ string be;
  char *ki = NULL, c;
  printf("Szoveg [max. 80 karakter]: "); gets(be);
  printf("Beszurando karakter: "); scanf("%c",&c);
  ki = KarBeszur(be, c);
  if (ki == NULL) {
    printf("Memoriafoglalas nem sikerult!\n"); fflush(stdin);
    getchar(); return 1;
  }
  printf("Modositott szoveg: \n"); SzovegKiir(ki);
  free(ki); fflush(stdin);getchar();
  return 0;
}
```

A *KarBeszur()* függvény a paraméterlistán átadott karaktertömb minden páros karaktere mögé másolja a *ch* karaktert, és a megváltozott tömb címét függvényértékként adja vissza.

```
char* KarBeszur(string szoveg, char ch)
{ char* kiszoveg = NULL;
  int i, j = -1, db;
  db = strlen(szoveg); /*a karaktersorozat karaktereinek száma */
  kiszoveg = (char*)malloc(2*db+1);
  if (kiszoveg != NULL)
  { /*a módosított szöveget és a sztringet záró '\0' tartalmazza */
    for (i=0; i<db; i++)
    {
      kiszoveg[++j] = szoveg[i];
      if (i % 2 == 0) kiszoveg[++j] = ch;
    }
    kiszoveg[++j] = '\0'; /* lezárja a sztringet */
  }
  return kiszoveg;
}
```

A *SzovegKiir()* függvény kiírja a paraméterlistán átadott karaktersorozatot.

```
void SzovegKiir(char *s)
{ int i;
  for (i = 0; *s;i++)
    printf("%c", *s++);  /* az s címét növeli */
  printf("\n");
}
```

A program futásának eredménye:

```
Szoveg [max. 80 karakter]: szalmakalap
Beszurando karakter: -
Modositott szoveg:
s-za-lm-ak-al-ap-
```

6.2.5 Változó hosszúságú paraméterlista

Írjunk programot, amely változó hosszúságú paraméterlista használatával kiszámítja a pozitív adatok összegét, valamint megszámlálja a negatív adatokat! Használjunk a számításokra kétféle megoldási módszert! (*ValtozoPar*)

```
#include <stdlib.h>
/* változó hosszúságú paraméterlistát kezelő függvények számára */
#include <stdarg.h>
int PozOsszeg(int elso, ...);
int NegativSzamlal(int n, ...);
int main()
{ int osszeg, ndb;
  osszeg = PozOsszeg(4,2,7,-1,-2,2,0);
  printf("Pozitiv adatok osszege: %d\n",osszeg);
  ndb = NegativSzamlal(6,-2,4,-5,8,-12,-8);
  printf("Negativ adatok szama: %d\n",ndb);
  getchar();
  return 0;
}
```

A *PozOsszeg()* függvény a változó hosszúságú paraméterlista azon hívását alkalmazza, amelyben az adatokat 0-val kell lezárni. A felsorolt adatok közül a pozitív adatok összegét adja vissza a **return** utasításban.

```
int PozOsszeg(int elso, ...)
{
  va_list t;
  int  osszeg = elso, kovetkezo;
  va_start(t, elso);
  while (kovetkezo = va_arg(t, int))
  {
     if (kovetkezo > 0) osszeg += kovetkezo;
  }
  va_end(t);
  return osszeg;
}
```

A *NegativSzamlal()* függvény a változó hosszúságú paraméterlista azon hívását alkalmazza, amelyben az első adat a paraméterlistán megadott adatok számát jelenti. A függvény a negatív adatok darabszámával tér vissza.

```
int NegativSzamlal(int n, ...)
{ va_list t;
  int i,  negdb = 0, kovetkezo;
  va_start(t, n);
  if (va_arg(t,int)< 0) negdb++;
  for (i = 1; i<n; i++)
  {
     kovetkezo = va_arg(t,int);
     if (kovetkezo < 0) negdb++;
  }
```

```
  va_end(t);
  return negdb;
}
```

A program futásának eredménye:

```
Pozitiv adatok osszege: 15
Negativ adatok szama: 4
```

6.2.6 A main() függvény paraméterei és visszatérési értéke

A ***main()*** függvény paraméterei:

```
int main( int argc, char *argv[])
```

argc – a tömbben található mutatók számát tárolja.
argv – egy karaktermutatókat tartalmazó tömbre mutat.

Megjegyzés: az *argc* értéke legalább 1, mivel az argv[0] mindig a program nevét tartalmazó sztringre mutat.

A ***main()*** függvény visszatérési értéke általában **int** típusú. A **return** mellett adjuk meg, hogy a program sikeresen (**return** 0;), illetve sikertelenül (**return** 1;) futott-e.

Írjunk programot, amelyben a ***main()*** függvény paramétereként egy szöveget és egy karaktert vár! A program jelezze, ha kevés paraméterrel indították! A program a megadott szöveg minden karaktere mögé másolja be a megadott karaktert! (*MainPar*)

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
 int i, j = -1;
 char buff1[20], kar[2], teljes[45];
 if (argc != 3)
 {
   printf("Hibas a parameterezes:\n");
   printf("A program inditasa: project1 szoveg1 szoveg2\n"); getchar();
   return 1;
 }
  strcpy(buff1, argv[1]); strcpy(kar, argv[2]);
  printf("Elso parameter   : %s\n",buff1);
  printf("Masodik parameter: %s\n",kar);
  for(i = 0; i<strlen(buff1); i++)
  {
     teljes[++j] = buff1[i];
     teljes[++j] = kar[0];
  }
  teljes[++j] = '\0';
  printf("Eredeti szoveg: %s\n",buff1);
  printf("Teljes szoveg: %s\n",teljes);
```

```
  fflush(stdin);getchar();
  return 0;
}
```

A lefordított program indítása:

mainpar almafa +

A program futásának eredménye:

```
Elso parameter    : almafa
Masodik parameter: +
Eredeti szoveg: almafa
Teljes szoveg: a+l+m+a+f+a+
```

6.2.7 Rekurzív függvények használata

Írjunk függvényt, amely pozitív valós alapot pozitív egész kitevőre emel *rekurzív* módon! (*Rekurzio*)

```
#include <stdio.h>
double hatvany(double alap, int kitevo);
int main()
{
  double a, ered;
  int k;
  printf("Hatvanyozas: pozitiv alap pozitiv kitevore\n");
  printf("Alap: "); scanf("%lf", &a);
  printf("Kitevo: "); scanf("%d",&k);
  ered = hatvany(a,k);
  if (ered > 0)
   printf("Hatvany erteke: %lf",ered);
  fflush(stdin);getchar();
  return 0;
}
```

A *hatvany()* függvény az *alap* és *kitevo* paramétereket ellenőrzi, és hibát jelez, ha az aktuális paraméter értéke nem megfelelő. A **switch** utasítás a *kitevo* alapján határozza meg a hatvány értékét. Ha a *kitevo* 0, a hatvány értéke 1, ha a *kitevo* 1, hatvány értéke az *alap*, egyébként (**default**) rekurzív módon önmagát hívja a függvény, csökkentve a kitevő értékét.

```
double hatvany(double alap, int kitevo)
{
  if (kitevo < 0 || alap < 0)
  {
    if (kitevo < 0) printf("A kitevo negativ!\n");
    if (alap < 0) printf("Az alap negativ!\n");
    return -1;
  }
```

```
  switch (kitevo)
  {
    case 0: return 1;
    case 1: return alap;
    default:
      return alap*hatvany(alap,kitevo-1);
  }
}
```

A program futásainak eredménye:

```
Hatvanyozas: pozitiv alap pozitiv kitevore
Alap: 2
Kitevo: 3
Hatvany erteke: 8.000000

Hatvanyozas: pozitiv alap pozitiv kitevore
Alap: 2
Kitevo: -3
A kitevo negativ!

Hatvanyozas: pozitiv alap pozitiv kitevore
Alap: -2
Kitevo: 3
Az alap negativ!
```

7. Modulok használata

A modulok használatával foglalkozunk ebben a fejezetben. Néhány példán keresztül bemutatjuk:

- az **extern** tárolási osztályú változók és függvények, valamint
- a modulok (több állományból felépülő alkalmazás) használatát.

7.1 Az extern tárolási osztály alkalmazása

Készítsünk többmodulos alkalmazást, amelyben **extern** tárolási osztályú függvények segítségével adattömböt csökkenő sorrendbe rendezzük! Használjuk a fejlesztőrendszer projektkezelési lehetőségeit! (*Modulok1*)

A függvényeket a *FUGGVENYEK.C* állományba tároljuk:

```
/*FUGGVENYEK.C*/
#include <stdio.h>
void TombOlvas(int* x, int* y, int n)
{ int i;
   for (i = 0; i< n; i++)
   {
      printf("\nx[%2d] = ",i); scanf("%d",&x[i]);
      printf("y[%2d] = ",i); scanf("%d",&y[i]);
   }
}
void CsokkenoRendezes(int* x, int* y, int n)
{ int i, j, mx, my;
  for (i = 0; i< n-1; i++)
  {
    for (j = i+1; j<n; j++)
    {
      if (x[j] > x[i])
      {
        mx = x[i]; x[i] = x[j]; x[j] = mx;
        my = y[i]; y[i] = y[j]; y[j] = my;
      }
    }
  }
}
void Megjelentes(int* x, int* y, int n)
{ int i;
  printf("\nRendezett adatok\n");
  printf("        x        y\n");
  printf("-------------------\n");
  for (i = 0; i<n; i++)
  {
    printf("%9d%9d\n",x[i],y[i]);
  }
}
```

A főfüggvény előtt az **extern** tárolási osztállyal kell megjelölnünk a használni kívánt függvények prototípusát.

```
/*MAIN.C*/
#include <stdio.h>
#include <stdlib.h>
extern void TombOlvas(int* x, int* y, int n);
extern void CsokkenoRendezes(int* x, int* y, int n);
extern void Megjelentes(int* x, int* y, int n);
int main()
{
  int *x1 = NULL, *y1 = NULL;
  int db;
  printf("Adatok szama: "); scanf("%d",&db);
  x1 = (int *)malloc(db * sizeof(int));
  y1 = (int *)malloc(db * sizeof(int));
  if (x1 == NULL || y1 == NULL)
  { printf("Memoriafoglalas nem sikerult!\n");
    fflush(stdin);getchar(); return 1;
  }
  TombOlvas(x1,y1,db);
  CsokkenoRendezes(x1,y1,db);
  Megjelentes(x1,y1,db);
  free(x1); free(y1);
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
Adatok szama: 3

x[ 0] = 1
y[ 0] = 3

x[ 1] = 7
y[ 1] = 4

x[ 2] = 2
y[ 2] = 8

Rendezett adatok
        x         y
--------------------
        7         4
        2         8
        1         3
```

7.2 Több modulból álló C program

Írjunk programot, amelynek főmodulja egy másik modulban tárolt tömbrendező függvényekhez deklarációs (.H) állomány segítségével fér hozzá! Az adatok rendezése növekvő sorrendben történjen! (*Modulok2*)

A függvények prototípusát egy deklarációs állományban (*.H*) ajánlott megadni.

```
/*FUGGVENYEK.H*/
void TombOlvas(int* x, int* y, int n);
void NovekvoRendezes(int* x, int* y, int n);
void Megjelentes(int* x, int* y, int n);
```

A függvényeket *.C* forrásfájlban dolgoztuk ki:

```
/*FUGGVENYEK.C*/
#include <stdio.h>
#include "Fuggvenyek.h"
void TombOlvas(int* x, int* y, int n)
{ int i;
   for (i = 0; i< n; i++)
   {
      printf("\nx[%2d] = ",i); scanf("%d",&x[i]);
      printf("y[%2d] = ",i); scanf("%d",&y[i]);
   }
}

void NovekvoRendezes(int* x, int* y, int n)
{
  int i, j;
  int mx, my;
  for (i = 0; i< n-1; i++)
  {
    for (j = i+1; j<n; j++)
    {
      if (x[j] < x[i])  /* rendezés növekvő sorrendben */
      {
        mx = x[i]; x[i] = x[j]; x[j] = mx;
        my = y[i]; y[i] = y[j]; y[j] = my;
      }
    }
  }
}

void Megjelentes(int* x, int* y, int n)
{
  int i;
  printf("\nRendezett adatok\n");
  printf("        x        y\n");
  printf("-------------------\n");
  for (i = 0; i<n; i++)
  {
    printf("%9d%9d\n",x[i],y[i]);  }
  }
}
```

A *MAIN.C* modul tartalmazza a főmodult:

```
/*MAIN.C*/
#include <stdio.h>
#include <stdlib.h>
#include "Fuggvenyek.h"
```

```
int main()
{
  int *x1 = NULL, *y1 = NULL, db;
  printf("Adatok szama: "); scanf("%d",&db);
  x1 = (int *)malloc(db * sizeof(int));
  y1 = (int *)malloc(db * sizeof(int));
  if (x1 == NULL || y1 == NULL)
  {
    printf("Memoriafoglalas nem sikerult!\n");
    fflush(stdin);getchar(); return 1;
  }
  TombOlvas(x1,y1,db);
  NovekvoRendezes(x1,y1,db);
  Megjelentes(x1,y1,db);
  free(x1);
  free(y1);
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
Adatok szama: 3

x[ 0] = 2
y[ 0] = 24

x[ 1] = 1
y[ 1] = 16

x[ 2] = 3
y[ 2] = 32

Rendezett adatok
        x         y
------------------
        3        32
        2        24
        1        16
```

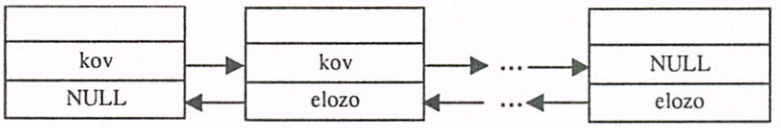
Megjegyzés: Az operációs rendszerek egy része különbséget tesz a fájlnevekben a kis- és a nagybetűk között. Erre ügyelnünk kell a saját include-állományaink nevénél!

8. Láncolt lista

Azonos típusú struktúrákból láncolt listát készíthetünk oly módon, hogy a listaelemeket alkotó struktúrát olyan pointerrel bővítjük, amely a struktúrára mutat. Ebben a fejezetben a bemutatjuk egy struktúra előre és hátra történő láncba szervezésének lépéseit, bővítését új struktúraelemmel, illetve a meglévő láncelemek törlését.

8.1 Előre-hátra láncolt lista

Írjunk programot, amely a *Henger* struktúrában tárolt sugár, magasság és a térfogat adatokat kétirányú láncba fűzi! Írjunk függvényeket, amelyek beolvassák az adatokat, elkészítik a láncot, kiírják a lánc tartalmát elölről és visszafelé, beszúrnak egy új elemet a lánc végére, valamint törölnek egy elemet a láncból! (*Lista*)



A *LISTAFG.H* állomány tartalmazza a *Henger* struktúra definícióját és a függvények prototípusát:

```
/*LISTAFG.H*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define PI 3.141592654
typedef struct hr {
                    int sugar;
                    int magassag;
                    double terfogat;
                    /* önhivatkozás típus névvel*/
                    struct hr *kov, *elozo;
                   }Henger;

Henger* Olvas(Henger **u);
void KiirElore(Henger* e);
void KiirHatra(Henger* u);
void BeszurV(Henger **e, Henger **v, int r, int h);
Henger* Torol(Henger *e, Henger **u, int x1, int y1);
```

A láncolt listák létrehozásához szükséges adatstruktúra esetében különös jelentősége van a típusnévnek (5.2.1 alfejezet), hiszen nélküle nem tudtuk volna definiálni az önhivatkozó adatstruktúrát.

Megjegyzendő, hogy az adatstruktúránk nemcsak lista, hanem ún. faszerkezet kialakítását is lehetővé teszi. Hogy listát vagy fát építünk-e, az attól függ, hogy hogyan szervezzük az adatokat a memóriában.

A *LISTAFG.CPP* állomány az alábbi függvényeket tartalmazza:

- Az *Olvas()* függvény prototípusa:

```
Henger* Olvas(Henger **u);
```

Az *Olvas()* függvény értékként adja vissza a lánc elejére, illetve paraméteren keresztül a lánc végére mutató pointert. Így elölről is és hátulról is könnyen hozzáférhetünk a lánc elmeihez.
Az *Olvas()* függvény beolvassa az adatok darabszámát, dinamikusan létrehozza a lánc elemeit a ***malloc()*** függvénnyel, majd beolvassa a *sugar* és a *magassag* adattagok értékét, kiszámítja a térfogatot, nullázza az *elozo* és a *kov* mutatókat, illetve láncba fűzi az elemeket. Figyeli, hogy volt-e már eleme a láncnak, mert ha még nem volt, akkor ez lesz az *elso*. A továbbiakban mindig a lánc végére fűzi az újabb elemeket.

- A *KiirElore()* függvény a paraméterben kapja a lánc első elemére mutató pointert. A függvény prototípusa:

```
void KiirElore(Henger* e);
```

A **while** ciklus figyeli, a lánc végét, amelyet a *NULL* értékű mutató jelez:

```
while (e != NULL)
```

A ciklusban tovább kell lépnünk a következő elemre, különben végtelen ciklust kapunk:

```
e = e->kov;
```

- A *KiirHatra()* függvény a paraméterlistán kapja a lánc utolsó elemére mutató pointert. A függvény prototípusa:

```
void KiirHatra(Henger* u);
```

Ebben az esetben a visszamutató pointert kell ez előző elemre állítani a láncon való végiglépkedés érdekében:

```
u = u->elozo;
```

- A *BeszurV()* függvény paraméterlistán kapja a lánc első elemére mutató pointert és a beszúrandó (*sugar, magassag*) adatokat. A függvény a lánc végére szúrja be az új listaelemet és kiszámítja a henger térfogatát. Visszatérési értékként a lánc elejére és végére mutató pointerek megváltozott értékét szolgáltatja.

```
void BeszurV(Henger **e, Henger **v, int r, int h);
```

- A *Torol()* függvény a paraméterlistán kapja a lánc *elso* és *utolso* elemére mutató pointert, valamint a törölni kívánt henger adatait. Az *utolso* mutató megváltozott értékét pointerben (***utolso*) adja vissza:

```
Henger *Torol(Henger *elso, Henger **utolso, int x1, int y1);
```

```
/*LISTAFG.C*/
#include "Listafg.h"
#include <process.h>  /* az exit() használatához */
Henger* Olvas(Henger **u)
{
 int i, db;
 Henger *uj, *elso = NULL, *utolso = NULL;
 printf("Adatok szama: ");
 scanf("%d",&db);
 for (i = 0; i< db; i++)
 {
   uj = (Henger*)malloc(sizeof(Henger));
   if (uj == NULL)
   {
    printf("Memoriafoglalas nem sikerult!\n");
    fflush(stdin);getchar(); exit(1); /* kilépés a programból */
   }
   printf("\nsugara: "); scanf("%d",&uj->sugar);
   printf("magassaga: "); scanf("%d",&uj->magassag);
   uj->terfogat = pow(uj->sugar,2)*PI*uj->magassag;
   uj->kov = NULL; uj->elozo = NULL;
   if (elso == NULL) elso = uj;
   else
   {
     utolso->kov = uj; uj->elozo = utolso;
   }
   utolso = uj;
 }
 *u = utolso;
 return elso;
}

void KiirElore(Henger* e)
{
  if (e != NULL)
  {
    printf("\nA lancolt lista elemei (elore)\n");
    printf("   sugar  magassag  terfogat    \n");
    while (e != NULL)
    {
      printf("%8d  %8d  %8.2lf\n",e->sugar,e->magassag,e->terfogat);
      e = e->kov;
    }
  }else printf("\nA lista ures\n");
}

void KiirHatra(Henger* u)
{ if (u != NULL)
  {
    printf("\nA lancolt lista elemei (visszafele)\n");
    printf("   sugar  magassag\n");
    while (u != NULL)
    {
      printf("%8d  %8d\n",u->sugar,u->magassag);
      u = u->elozo;
    }
```

```
  }else printf("\nA lista ures\n");
}

void BeszurV(Henger **e, Henger **v, int r, int h)
{
   Henger *uj;
   Henger *elso, *utolso;
   elso = *e;
   utolso = *v;
   uj = (Henger*)malloc(sizeof(Henger));
   if (uj == NULL)
   {
    printf("Memoriafoglalas nem sikerult!\n");
    fflush(stdin);getchar(); exit(1); /* kilépés a programból */
   }
   uj->sugar = r;
   uj->magassag = h;
   uj->terfogat = pow(uj->sugar,2)*PI*uj->magassag;
   uj->kov = NULL;
   uj->elozo = NULL;
   if (elso == NULL && utolso == NULL )
   {
      *e = uj; *v = uj;
   }
   else
   {
     elso->elozo = uj;
     uj->kov = elso;
     *e = uj;
     *v = utolso;
   }
}

Henger* Torol(Henger *elso, Henger **utolso, int r, int h)
{
  Henger *p, *pkov;
  int talalt = 0;
  p = elso;
  while (p!= NULL)
  {
    if (p->sugar == r && p->magassag == h)
    {
      talalt++;
      if (p->elozo == NULL && p->kov == NULL)
      {
        elso = NULL; *utolso = NULL; free(p); break;
      }
      if (p->elozo != NULL) (p->elozo)->kov = p->kov;
      else elso = p->kov;
      if (p->kov != NULL) (p->kov)->elozo = p->elozo;
      else *utolso = p->elozo;
      free(p); break;
    }
    p = p->kov;
  }
```

```
  if (talalt == 0)
    printf("\nsugar: %d magassag: %d nincs a listaban'\n",r,h);
  return elso;
}
```

A *main()* függvény bemutatja a *Henger* struktúrából felépített lánc módosítását, új elem beszúrását, listázását és törlését.

```
/*MAIN.C*/
#include "Listafg.h"
int main()
{
  int i, r, h, tdb;
  Henger *pelso = NULL, *putolso = NULL, *p, *p1;
  pelso = Olvas(&putolso);
  KiirElore(pelso);
  printf("\nKonstans beszuras\n");
  BeszurV(&pelso,&putolso,1,1);  /* beszúrás a láncba */
  KiirHatra(putolso);
  printf("\nTorlesek szama: "); scanf("%d",&tdb);

  for(i = 1; i<=tdb; i++)
  { printf("\nHenger torlese\n");
    printf(" sugar   : "); scanf("%d",&r);
    printf(" magassag: "); scanf("%d",&h);
    pelso = Torol(pelso, &putolso, r,h); /* törlés a láncból */
    KiirElore(pelso);
  }

  p = pelso;
  while (p != NULL) /* a lista elemeinek felszabadítása */
  {
    p1 = p->kov; free(p); p = p1;
  }
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
Adatok szama: 2

sugara: 3
magassaga: 4

sugara: 6
magassaga: 7

A lancolt lista elemei (elore)
   sugar  magassag  terfogat
       3         4    113.10
       6         7    791.68
```

```
Konstans beszuras

A lancolt lista elemei (visszafele)
   sugar  magassag
       6         7
       3         4
       1         1

Torlesek szama: 1

Henger torlese
 sugar   : 6
 magassag: 7

A lancolt lista elemei (elore)
   sugar  magassag  terfogat
       1         1      3.14
       3         4    113.10
```

9. Algoritmusok programozása

Az algoritmusok programozására mutatunk be néhány példát: osztók, legnagyobb közös osztó, legkisebb közös többszörös meghatározása, keresés és rendezés. A feladatokat projekt felhasználásával oldjuk meg.

9.1 Osztók, legnagyobb közös osztó, legkisebb közös többszörös meghatározása

Olvassunk be kettőnél nagyobb egész számot, és írjuk ki a szám összes osztóját, jelezzük, ha prímszám! Oldjuk meg a feladatot projekt használatával háromféleképpen: **for**, **while** és **do-while** ciklusokkal! (*Alg1*)

```
/*FUGGV.H*/
#include <stdio.h>
void oszto1(long int szam);
void oszto2(long int szam);
void oszto3(long int szam);

/*FUGGV.C*/
#include "Fuggv.h"
void oszto1(long int szam) /* for ciklussal */
{
  long int oszto, fele, van = 0;
  fele = szam /2;
  printf("Osztoi:\n");
  for (oszto = 2; oszto <= fele; oszto++)
  {
    if (szam % oszto == 0)
    { printf("%8d",oszto); van++; }
  }
  if (!van) printf("  Primszam!\n");
}

void oszto2(long int szam)  /* while ciklussal */
{
  long int oszto, fele, van = 0;
  fele = szam /2;
  printf("\nOsztoi:\n");
  oszto = 2;
  while (oszto <= fele)
  {
    if (szam % oszto == 0)
    { printf("%8d",oszto); van++; }
    oszto++;
  }
  if (!van) printf("  Primszam!\n");
}
```

```
void oszto3(long int szam)
{
  long int oszto, fele, van = 0;
  fele = szam /2;
  printf("\nOsztoi:\n");
  oszto = 2;
  do                          /* do-while ciklussal */
  {
    if (szam % oszto == 0)
    { printf("%8d",oszto); van++; }
    oszto++;
  } while(oszto <= fele);
  if (!van) printf("  Primszam!\n");
}

/*MAIN.C*/
#include "Fuggv.h"
int main()
{
  int x;
  do{
   printf("Szam (>2):  "); scanf("%d",&x);
  }while (x <=2);
  printf("\nfor ciklussal\n");
  oszto1(x);
  printf("\nFolytatas Enter leutesevel...\n\n");
  fflush(stdin);getchar();
  printf("while ciklussal");
  oszto2(x);
  printf("\nFolytatas Enter leutesevel...\n\n");
  fflush(stdin);getchar();
  printf("do - while ciklussal");
  oszto3(x);
  printf("\nFolytatas Enter leutesevel...\n\n");
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
Szam (>2):  128

for ciklussal
Osztoi:
       2       4       8      16      32      64
Folytatas az Enter leutesevel...

while ciklussal
Osztoi:
       2       4       8      16      32      64
Folytatas az Enter leutesevel...

do - while ciklussal
Osztoi:
       2       4       8      16      32      64
Folytatas az Enter leutesevel...
```

Olvassunk be kettőnél nagyobb egész számot, határozzuk meg a szám törzstényezőit, és jelenítsük meg hatványkitevős alakban! A feladatot oldjuk meg projekt felhasználásával! (*Alg2*)

```
/*FUGGV1.H*/
#include <stdio.h>
void torzstenyezok(int szam);
void hatvanykitevos(int szam);

/*FUGGV1.C*/
#include "Fuggv1.h"
/* a szam törzstényzőkre való bontása */
void torzstenyezok(int szam)
{
  int oszto;
  if (szam != 1 && szam > 0)
  {
    printf("\nTorzstenyezok:\n");
    oszto = 2;
    while (szam != 1 && szam > 0)
    {
      while (szam %oszto != 0)
      {
        oszto++;
      }
      szam /= oszto;
      printf("%d",oszto);
      if (szam != 1) printf(" * ");
    }
  }
}

/* a szam felbontása hatványkitevős alakra */
void hatvanykitevos(int szam)
{
  int oszto = 2, hatv = 1, elozo = 1;
  if (szam != 1 && szam > 0)
  {
    printf("\n\nHatvanykitevos alak:\n");
    while (szam != 1 && szam > 0)
    {
      while (szam % oszto != 0)
      {
        oszto++;
      }
      szam /= oszto;
      if (oszto == elozo)
        hatv++;
      else
      {
        if (elozo != 1)
        {
          if (hatv > 1)
```

```
            printf("%d^%d*",elozo, hatv);
          else
            printf("%d*",elozo);
        }
        hatv = 1;
      }
      elozo = oszto;
      if (szam == 1)
       if (hatv > 1)
         printf("%d^%d",oszto,hatv);
       else
         printf("%d",oszto);
    }
  }
}

/*MAIN.C*/
#include "Fuggv1.h"
int main()
{
  int x;
  do{
     printf("Szam: "); scanf("%d",&x);
  }while (x <=2);
  torzstenyezok(x);
  hatvanykitevos(x);
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
Szam: 100000

Torzstenyezok:
2 * 2 * 2 * 2 * 2 * 5 * 5 * 5 * 5 * 5

Hatvanykitevos alak:
2^5*5^5
```

Olvassunk be két pozitív egész számot, határozzuk meg a legnagyobb közös osztójukat és a legkisebb közös többszörösüket! A megoldást projekt használatával készítsük el! (*Alg3*)

A legnagyobb közös osztót és a legkisebb közös többszöröst kétféle határozzuk meg:

- A *végigpróbál* módszer. 2-től a kisebb számig megnézi mindkét szám közös osztóit.
- A *törzstényezőkre* bontás módszere. A legkisebb közös többszörös meghatározásánál törzstényezőkre bontjuk a számokat. Először az *szam1*-nek keressük a törzstényezőit, a megtalált törzstényezővel elosztjuk a *szam1*-t. A *szám2*-őt csak akkor osztjuk el, ha osztója a számnak. A legkisebb többszöröst beszorozzuk a

megtalált törzstényezővel, ezt addig végezzük, míg a *szam1* 1 lesz. Ezután a *szam2*-vel is addig folytatjuk a törzstényezőkre bontást, míg a *szam2* is 1-gyé válik.

```
/*FUGGV3*/
/* az algoritmusok függvényeinek prototípusai */
#include <stdio.h>
void Vegigprobal(int szam1, int szam2);
void Torzstenyezok_lnko(int szam1, int szam2);

/*FUGGV3.C*/          /* algoritmusok függvényei */
#include "Fuggv3.h"
void Vegigprobal(int szam1, int szam2)
{
  int eddig, oszto, kozos =1, v;
  printf("\nVegigprobal modszerrel\n\n");
  if (szam1 > 0 && szam2 > 0)
  {
     eddig = szam1;
     if (szam2 < eddig)
        eddig = szam2;
    for (oszto = 2; oszto <= eddig; oszto++)
    {
      if (szam1 % oszto == 0 && szam2 % oszto == 0)
         kozos = oszto;
    }
    printf("Legnagyobb kozos oszto: %d\n",kozos);
    v = szam1*szam2/kozos;
    printf("Legkisebb kozos tobbszoros: %d",v);
  }
}

void Torzstenyezok_lnko(int szam1, int szam2)
{
  int s1, s2, lktobb, oszto1, oszto2, v;
  printf("\n\nTorzstenyezokre bontas modszerrel\n\n");
  if (szam1 > 0 && szam2 > 0)
  {
    s1 = szam1;
    s2 = szam2;
    lktobb = 1;
    while (szam2 != 1 || szam1 != 1)
    {
      oszto1 = 2;
      oszto2 = 2;
      while (szam1 != 1)
      {
        while (szam1 % oszto1 != 0)
        {
          oszto1++;
        }
        if (szam2 % oszto1 == 0)
          szam2 /= oszto1;
        lktobb*= oszto1;
```

```
        szam1 /= oszto1;
      }
      while (szam2 != 1)
      {
        while (szam2 % oszto2 != 0)
        {
          oszto2++;
        }
        szam2 /=oszto2;
        lktobb*=oszto2;
      }
    }
    v = s1*s2/lktobb;
    printf("Legnagyobb kozos oszto: %d\n",v);
    printf("Legkisebb kozos tobbszoros: %d",lktobb);
  }
}
```

```
/*MAIN.C*/
#include "Fuggv3.h"
int main()
{
  int x, y;
  do{
    printf("1. Szam: "); scanf("%d",&x);
  }while (x < 2);
  do{
    printf("2. Szam: "); scanf("%d",&y);
  }while (x < 2);
  Vegigprobal(x,y);
  Torzstenyezok_lnko(x,y);
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
1. Szam: 8
2. Szam: 12

Vegigprobal modszerrel

Legnagyobb kozos oszto: 4
Legkisebb kozos tobbszoros: 24

Torzstenyezokre bontas modszerrel

Legnagyobb kozos oszto: 4
Legkisebb kozos tobbszoros: 24
```

9.2 Keresési és rendezési algoritmusok

A rendezés meggyorsítja az elemek későbbi keresését, mivel egy rendezett halmazon való keresés jóval gyorsabban elvégezhető, mint a rendezetlenben. Ezért a rendezés kiemelkedően fontos tevékenység az adatfeldolgozás során. A rendezésnek az a feladata, hogy később megkönnyítse az elemek keresését.

9.2.1 Keresési algoritmusok

Az alábbiakban két keresési módszert nézünk meg részletesebben:
- szekvenciális,
- bináris.

Szekvenciális és bináris keresést végző függvények a megtalált adat tömbbeli indexét adják vissza, vagy nemlétező adat esetén *"hibás adat!"* hibajelzést küldenek.

```
/*KERESES*/
/* szekvenciális és bináris keresés  */
#include <stdio.h>
#define DB 50
typedef int itype[DB+1];
typedef int ibuff;

void olvas_int(itype x, int *m);
int seq_keres(itype, int, ibuff);
int bin_keres(itype, int, ibuff );
int main()
{
  itype y;
  int   n, j, k, t;
  ibuff adat;
  olvas_int(y,&n);
  printf("\nSzekvencialis kereses\n");
  printf("Kereses szama: "); scanf("%d",&k);
  for (j = 1; j<=k; j++)
  {
    printf("adat : "); scanf("%d",&adat);
    t = seq_keres(y,n,adat);
    if(t >= 0)
      printf("%d helye %d\n",adat,t);
    else printf("%d hibas adat!\n",adat);
  }
  printf("\nBinaris kereses\n");
  printf("Kereses szama: "); scanf("%d",&k);
  for (j = 1; j <= k; j++ )
  {
    printf("adat : "); scanf("%d", &adat);
    t = seq_keres(y,n,adat);
    if (t >= 0)
      printf("%d helye %d\n",adat,t);
```

```
    else printf("%d hibas adat!\n",adat);
    }
    fflush(stdin);
    getchar();
    return 0;
}

void olvas_int(itype x, int *m)
{
  int i;
  printf("Egesz tipusu adatok beolvasasa\n");
  do
  {
    printf("Az adatok szama : "); scanf("%d",m);
    if (*m > DB) printf("Az adatok max. szama : %d ",DB);
  }while (*m < 2 || *m > DB);

  for (i = 0; i < *m; i++)
  {
    printf(" [%2d] = ",i); scanf("%d",&x[i]);
  }
}

int seq_keres(itype x, int m, ibuff adat)
{
  int i = 0;
  while ( adat != x[i]  &&  i < m ) { i = i+1; }
  if (i >= m) return -1;
  else return i;
}

int bin_keres(itype x, int m, ibuff adat)
{ int  a,f,k, talal;
   a = 0; f = m;
   talal = 0;
   while ( a < f && ! talal)
   {
     k = (a+f) / 2;
     if (adat < x[k]) f = k-1;
       else
     if (adat > x[k])  a = k+1;
       else talal = 1; /* megtalálta */
   }
   if (talal) return k;
      else return -1;  /* nem talált */
} /* bin_keres  */
```

A program futásának eredménye:

```
Egesz tipusu adatok beolvasasa
Az adatok szama : 5
 [ 0] = 24
 [ 1] = 5
 [ 2] = 65
 [ 3] = 71
 [ 4] = 43
```

```
Szekvencialis kereses
Kereses szama: 2
adat : 65
65 helye 2
adat : 71
71 helye 3

Binaris kereses
Kereses szama: 2
adat : 11
11 hibas adat!
adat : 43
43 helye 4
```

9.2.2 Rendezési algoritmusok

Rendezésnek nevezzük azt a folyamatot, amikor egy halmaz elemeit valamilyen szabály szerint sorba állítjuk. Sokfajta rendező algoritmus létezik, amelyek ugyanazt a feladatot más-más módon oldják meg.

- ***Rendezés cserével***

Legyen az adatok száma *n*. A feladat az adatokat növekvő sorrendbe rendezése.

```
void csere(tomb a, int n)
{
  int i,j,s;
  for (i = 0; i<n-1; i++)
  {
    for (j = i+1; j<n; j++)
    {
      if (a[i] > a[j])
      {
        s = a[i];
        a[i] = a[j];
        a[j] = s;
      }
    }
  }
}
```

ciklus		index					
i	j	1	2	3	4	5	
0	1	**5**	**2**	3	1	4	
	2	**2**	5	**3**	1	4	nincs csere
	3	**2**	5	3	**1**	4	
	4	**1**	5	3	2	**4**	nincs csere
1	2	1	**5**	**3**	2	4	
	3	1	**3**	5	**2**	4	
	4	1	**2**	5	3	**4**	nincs csere
2	3	1	2	**5**	**3**	4	
	4	1	2	**3**	5	**4**	nincs csere
3	4	1	2	3	**5**	**4**	

Rendezett adatsor: 1, 2, 3, 4, 5.

A rögzített *i*-edik elemet összehasonlítjuk a nála nagyobb indexű elemekkel. Ha a vizsgált elem kisebb az *i*-edik elemnél, az elemeket felcseréljük egymással. A fenti algoritmust két egymásba ágyazott ciklussal valósítjuk meg.

- ***Rendezés közvetlen beszúrással***

A közvetlen beszúrással történő rendezés esetén az algoritmus úgy működik, hogy az első elemtől kezdve minden lépésben kiemeljük a sorozat előző (kisebb) elemét és beszúrjuk a megfelelő helyre.

```
void kozvetlen(tomb a, int n)
{
  int i,j,buf;
  for (i = 1; i < n; i++)
  {
    buf = a[i];
    j = i-1;
    while (j >= 0 && buf < a[j])
    {
      a[j+1] = a[j];
      j--;
    }
    a[j+1] = buf;
  }
}
```

Adatsor: 5, 2, 3, 1, 4.

lépés					
0	5	**2**	3	1	4
1	2	5	**3**	1	4
2	2	3	5	**1**	4
3	1	2	3	5	**4**

Rendezett adatsor: 1, 2, 3, 4, 5.

- ***Buborékrendezés***

A rendezés elnevezése abból ered, hogy a rendezendő adatokat a víztartályban lebegő buborékoknak képzeljük, és a rendezés minden lépésében a még nem rendezett legkisebb buborék felemelkedik a súlyának megfelelő szintre.

```
void buborek(tomb a, int n)
{
 int i,j, t;
 for (i = 1; i<n; i++)
   for (j = i-1; j>=0 && a[j]>a[j+1]; j--)
   {
      t = a[j];
      a[j] = a[j+1];
      a[j+1] = t;
   }
}
```

Adatsor: 5, 2, 3, 1, 4.

lépés					
0	5	2	3	**1**	4
1	1	5	**2**	3	4
2	1	2	5	**3**	4
3	1	2	3	5	**4**

Rendezett adatsor: 1, 2, 3, 4, 5.

Ennél az algoritmusnál előfordulhat, hogy az adatok rendezése az adatok számával megegyező lépésszámnál korábban befejeződik. Ezért az "üres" lépések kiküszöbölésére minden lépésben feljegyezzük, történt-e csere vagy sem. Ha nem, akkor ez azt jelenti, hogy az összes elem a helyén van, és az algoritmus befejeződhet. Az algoritmus tovább javítható, ha megjegyezzük az utolsó csere indexét, és csak a nála nagyobb indexű elemeket vizsgáljuk (a többi elem már jó sorrendben áll).

- ***Shell-rendezés***

Donald Shell 1959-ben talált egy jobb rendezési módot, amelyet róla neveztek el. A módszer lényege, hogy az egymástól távolabb elhelyezkedő adatokat hasonlítunk össze, így azok hamarabb kerülnek a saját rendezett helyükre. A Shell-rendezés csak a különböző lépésközzel végzett feldolgozásban különbözik a beszúrással való rendezéstől.

```
void shell(tomb a, int n)
{
 int i,j, t, k;
 for (k = n/2; k>0; k = k/2)
 {
   for (i = k; i<n; i++)
   {

     for (j = i-k; j>=0 && a[j]>a[j+k]; j -= k)
     {
       t = a[j];
       a[j] = a[j+k];
       a[j+k] = t;
     }
    }
  }
}
```

Adatsor: 5, 2, 3, 1, 4.

lépés					
0	**5**	2	**3**	1	4
1	3	**2**	5	**1**	4
2	3	1	**5**	2	**4**
3	3	**1**	4	2	5
4	1	3	**4**	**2**	5
5	1	2	3	4	**5**

Rendezett adatsor: 1, 2, 3, 4, 5.

```
/*RENDEZES*/  /*a rendező algoritmusok aktiválása */
#include <stdio.h>
#include <stdlib.h>
typedef int tomb[20];
void menu(int *sorszam);
void olvas(tomb a, int *n);
void kiir(tomb a, int n);
void buborek(tomb a, int n);
void csere(tomb a, int n);
void kozvetlen(tomb a, int n);
void shell(tomb a, int n);
int main()
{ tomb  x;
  int   db ,sor;
    menu(&sor);
    switch (sor)
    {
      case 1: olvas(x,&db); csere(x,db);
              printf("\nCsere modszer\n"); kiir(x,db);
              break;
      case 2: olvas(x,&db); kozvetlen(x,db);
              printf("\nKozvetlen beszuras\n"); kiir(x,db);
              break;
      case 3: olvas(x,&db); buborek(x,db);
              printf("\nBuborek modszer\n"); kiir(x,db);
              break;
      case 4: olvas(x,&db); shell(x,db);
              printf("\nShell modszer\n"); kiir(x,db);
              break;
      case 5: ;
   }
   fflush(stdin);getchar();
   return 0;
}
```

```
void menu(int *sorszam)
{
  int kod;
  do {
      printf("      Rendezes modszerei \n");
      printf(" 1.  Csere\n");
      printf(" 2.  Kozvetlen beszuras\n");
      printf(" 3.  Buborek\n");
      printf(" 4.  Shell\n");
      printf(" 5.  Kilepes\n");
      printf("\nA modszer sorszama: ");
      scanf("%d",&kod);
    }while (kod <1 || kod >5);
    *sorszam = kod;
    printf("\n");
}
void olvas(tomb a, int *n)
{ int   i, db;
  do
  {
    printf("Elemek szama (max. 20): "); scanf("%d",n);
  }while (*n < 2 || *n >20);
  for (i = 0; i<*n; i++)
  {
    printf("%2d. adat = ",i); scanf("%d",&a[i]);
  }
}
void kiir(tomb a, int n)
{ int i;
  for (i = 0; i<n ; i++)
  {
    printf("[%2d] = %d\n",i,a[i]);
  }
}
```

- ***Gyorsrendezés (Quicksort)***

Az alapváltozatot, amely csere elven alapszik 1962-ben C.A.R.Hoare hozta nyilvánosságra. Elve a következő: Kiválasztjuk a tömb középső elemét (felezzük a tömböt). Balról megkeressük azt az első elemet, amely ennél nem kisebb, jobbról azt az első elemet, amely ennél nem nagyobb. Kicseréljük a két elemet, és folytatjuk a cserélgetést, amíg a bal oldalon a középső elemnél (amely a cserélgetés közben természetesen nem cserélődhet) nem nagyobb, jobb oldalon pedig mind kisebb elemek állnak. Ezt követően rekurzív hívással rendezzük a tömb alsó és felső felét és így tovább.

```
/*QUICKREND*/
/* int tipusú adatok rendezése quick módszerrel */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define DB  50
```

```
typedef int adat[DB+1];
void olvas_i(adat x, int *m);
void kiir_i(adat x, int m);
void qs(int l, int r, adat xx);
void qs_i(adat x, int m);
int main()
{ adat y, r;
  int  n;
  printf("Egesz tipusu adatok rendezese\n");
  olvas_i(y,&n);
  qs_i(y,n);
  printf("\nRendezett adatok \n"); kiir_i(y,n);
  fflush(stdin);getchar();
  return 0;
}

void olvas_i(adat x, int *m)
{ int i;
  printf("\nRendezendo adatok\n");
  do
  {
    printf("Az adatok szama (max.%d): ",DB); scanf("%d",m);
    if (*m > DB) printf("Az adatok max. szama : %d\n",DB);
  }while (*m >= DB);
  for (i = 1; i<= *m; i++)
  {
    printf("%2d. adat : ",i); scanf("%d",&x[i]);
  }
}

void kiir_i(adat x, int m)
{ int i;
  printf("Az adatok szama : %d\n",m);
  for (i = 1; i<= m; i++)
  {
    printf(" [%2d]= %d\n",i,x[i]);
  }
}

void qs(int l,int r, adat xx)
{ int  i,j;
  int  buf1,buf2;
  i = l; j = r;
  buf1 = xx[(l+r) / 2];
  do
  {
     while (xx[i] < buf1) i++;
     while (buf1  < xx[j]) j--;
     if (i <= j)
     {
          buf2  = xx[i];
          xx[i] = xx[j];
          xx[j] = buf2;
          i++; j--;
     }
   }while (i <= j);
```

```
   if ( l < j )  qs(l,j,xx);
   if ( l < r )   qs(i,r,xx);
}

void qs_i(adat x, int m)
{
  qs(1,m,x);
}
```

A program futásának eredménye:

```
Egesz tipusu adatok rendezes

Rendezendo adatok
Az adatok szama (max.50): 5
 1. adat : 12
 2. adat : 67
 3. adat : 32
 4. adat : 2
 5. adat : 18

Rendezett adatok
Az adatok szama : 5
 [ 1]= 2
 [ 2]= 12
 [ 3]= 18
 [ 4]= 32
 [ 5]= 67
```

- ***Rendezés fastruktúrával***

Valós számok rendezése fastruktúra felhasználásával.

```
/*FAREND*/
#include <stdlib.h>
#include <stdio.h>

/* A fában tárolt adatelemek típusa */

typedef struct treenode
{
  double           data;
  struct treenode *left;
  struct treenode *right;
} TREENODE;
typedef TREENODE *TREE;

/*  A csomópontok kezelése:
    tn_create  - új elem létrehozása
    tn_destroy - elem megsemmisítése
*/
```

```
TREENODE *tn_create(double value)
{
  TREENODE *nptr = (TREENODE*)malloc(sizeof(TREENODE));
  if (nptr != NULL)
  {
    nptr->data = value;
    nptr->left = nptr->right = NULL;
  }
  return nptr;
}

void tn_destroy(TREENODE *ptr)
{
  free(ptr);
}

/*  A bináris keresőfa kezelése:
    tree_insert   - elem beillesztése a fába
    tree_destroy  - elem levétele a fából
    tree_traverse - a fa rendezett bejárása a kijelölt művelet
                    elvégzésével
*/

 #define tree_create()  NULL            /* üres fa létrehozása */

TREE tree_insert(TREE root, double value)
{
  TREENODE *curr = root;                    /* az aktuális elem */
  TREENODE *prev = NULL;                    /* a szülő elem */
  TREENODE *temp = tn_create(value);        /* új elem */
  int cmp = 0;                              /* a hasonlítás eredménye */
  for (; curr != NULL; curr = cmp ? curr->left : curr->right)
  {
    cmp  = value < curr->data;
    prev = curr;
  }
  if (prev == NULL)                         /* az elem a fa szülője */
    root = temp;
  else if (cmp)
    prev->left = temp;                      /* bal gyermek */

  else
    prev->right = temp;                     /* jobb gyermek */
  return root;
}

void tree_destroy(TREE root)
{
  if (root != NULL)
  {
    tree_destroy(root->left);
    tree_destroy(root->right);
    tn_destroy(root);
  }
}
```

```
void tree_print(TREE root)
{
  if (root != NULL)
  {
    tree_print(root->left);
    printf("%8.3lf\n",root->data);
    tree_print(root->right);
  }
}

/* A főprogram */
int main()
{
  TREE st = tree_create();
  double d;
  int i, n;
  printf("\nAz adatok szama: ");
  scanf("%d",&n);

   /* A rendezett fa felépítése */
  for (i = 0; i < n; i++)
  {
     printf("%2d. adat = ",i);
     scanf("%lf",&d);
     st = tree_insert(st, d);
  }

  if (n > 0) printf("\nA rendezett adatok:\n");
  /* A bejárása az elemek kiírásával */
  tree_print(st);
 /* A fa megszüntetése */
  tree_destroy(st);
  fflush(stdin); getchar();
  return 0;
}
```

A program futásának eredménye:

```
Az adatok szama: 4
 0. adat = 5
 1. adat = 1
 2. adat = 7
 3. adat = 2

A rendezett adatok:
   1.000
   2.000
   5.000
   7.000
```

10. Szövegfájlok kezelése

Szöveges állományba való írásra és olvasásra mutatunk be különféle feladatokat ebben a fejezetben. Előtte azonban röviden áttekintjük a fájlkezeléssel kapcsolatos elméletet.

- **Állományok típusai**

  Szöveges állományok

  A szöveges állomány tetszőleges hosszúságú sorokból épül fel, a sor végét '\n' zárja. A '\n' karakter értelmezése az operációs rendszertől függ. A Windows rendszereknél, szöveges állományok esetében a programbeli '\n' karakter a CR/LF (kocsi vissza, soremelés) karaktereknek felel meg, a Unix (Linux) operációs rendszerek esetében pedig az LF karakternek. A szöveges állományok kezelésnél a '\n' karakter automatikusan konvertálódik az operációs rendszernek megfelelő fizikai reprezentációba. Az ún. bináris módban megnyitott adatfolyamoknál ez a konverzió elmarad. Az állomány végét az EOF (**E**nd **O**f **F**ile) vezérlőkarakter zárhatja.

  Bináris állományok

  A bináris állományok bájtokból épülnek fel.

- **Állományok kezelésének általános leírása**

  Az állományok tartalmának eléréséhez mindig ugyanazokat a lépéseket kell végrehajtani:

  - a fájlt azonosító mutató definiálása,
  - az állomány megnyitása,
  - az állomány tartalmának feldolgozása fájlműveletek (olvasás, írás, pozicionálás stb.) felhasználásával.
  - A műveletek elvégzése után az állomány lezárása.

  Fájlmutató definiálása

  A programban a felhasználni kívánt adatállományhoz külön fájlmutatót kell létrehoznunk. A ***FILE**** típusú fájlmutató – melynek deklarációja az *STDIO.H* állományban található – azonosítja az állományt.

  Az állomány lezárása

  Az írásra és olvasásra megnyitott állományokat a használat végén az ***fclose()*** függvény hívásával le kell zárnunk.

  Ha szöveges állományon írási műveletet hajtottunk létre, akkor az ***fflush()*** függvénnyel a memóriában található átmeneti pufferek tartalmát lezárás előtt az állományba kell írnunk.

Mindkét függvénynek argumentuma a fájlmutató:

```
FILE *f;
  . . .
fflush(f);
fclose(f);
```

Az állomány végének vizsgálata

Az állomány olvasásakor az *feof()* függvény 1 értéke jelzi az állomány vége eseményt.

```
int feof(FILE *f);
```

Az állomány végének figyelését általában **while** ciklusban végezzük:

```
while (!feof(f))
{
   /* az állomány olvasása */
}
fclose(f);
```

- **Szöveges állomány használata**

A szöveges állomány megnyitása

A háttértáron lévő állomány tartalmához csak a fájl megnyitása után férhetünk hozzá. A megnyitás hozza létre a kapcsolatot a fájlmutató és az operációs rendszer állománya között. A nyitáskor meg kell adnunk az állomány típusát és a hozzáférés módját:

```
FILE *fopen(const char *fajlnev, const char *mod);
```

A *fajlnev* paraméter helyén adjuk meg az állomány operációs rendszerben használt nevét, melyben a teljes elérési útvonal is szerepelhet.
A *mod* paraméter szintén sztring, amely a fájl elérését és típusát határozza meg:

Szöveges	*A **mod** paraméter jelentése*
`"r", "rt"`	Létező állomány megnyitása.
`"w", "wt"`	Új állomány megnyitása. Ha már létezik, akkor a tartalma elvész.
`"a", "at"`	Az állomány megnyitása a végéhez való hozzáírásra, tehát a fájl bővítése. Ha az állomány nem létezik, akkor nyitáskor létrejön.
`"r+", "r+t"`	A létező állomány megnyitása írásra és olvasásra (*update*).
`"w+", "w+t"`	Új állomány megnyitása írásra és olvasásra (*update*). Ha a fájl már létezik, akkor a tartalma elvész.
`"a+", "a+t"`	Az állomány megnyitása a végén végzett írásra és olvasásra (*update*). Ha a fájl nem létezik, akkor most jön létre.

Ha nem sikerült az állomány megnyitása, akkor adjunk hibajelzést!

```
if ((f1 = fopen("mondat.txt","r")) == NULL)
{
   fprintf(stderr,"A mondat.txt allomany nem letezik!\n");
   ...
}
```

Karakter olvasása és írása

Karakter olvasása:

```
int fgetc(FILE *adatfolyam);
```

Karakter írása:

```
int fputc(int c, FILE *adatfolyam);
```

Szóközt tartalmazó karaktersorozat olvasása és írása

Egy sort olvas, amely szóközt is tartalmazhat:

```
char *fgets(char *s, int n, FILE *adatfolyam);
```

Egy sort ír, amely szóközt is tartalmazhat:

```
int fputs(const char *s, FILE * adatfolyam);
```

Formázott adatok olvasása és írása

A ***scanf()*** és ***printf()*** függvényekhez hasonlóan formázott adatokat olvashatunk állományból és írhatunk állományba:

```
int fscanf(FILE * adatfolyam, const char *formatum, ...);
int fprintf(FILE * adatfolyam, const char *formatum, ...);
```

10.1 Szöveges állományba való írás és olvasás

Íjunk programot, amely az *UTVONAL.TXT* szöveges állományból beolvassa az *Uticel* típusú struktúra adattagjait! Listázzuk ki azon autók adatait, melyek 100 és 200 km közötti távolságra mentek! Keressük ki a legtávolabbra és a legközelebbre menő járműveket, adjuk össze az autók által megtett összes km-t, valamint írjuk ki az utolsó olyan gépkocsi típusát, amely Esztergomba ment! (*TextFile1*)

Használja fel az alábbi definíciót és deklarációt!

```
typedef struct u {
                  char kocsitipus[12];
                  int km;
                  char varos[12];
                }Uticel;
```

```
int main()
{
  Uticel ut, LegT, LegK;
  char kocsitip[13];
  int ossz_km = 0;
  FILE *inp;
  kocsitip[0] = ' ';
```

A formátumsztring az adattagok beolvasására:

```
"%12s%4d  %12s\n", ut.kocsitipus,&ut.km,ut.varos
```

Az *UTVONAL.TXT* szöveges állomány tartalma:

```
Skoda         217  Sopron
Mercedes      106  Siofok
Opel          46   Esztergom
Suzuki        179  Miskolcs
Ford          245  Nyiregyhaza
```

A feladat kidolgozása során ügyeljünk arra, hogy két sztringet (karaktersorozatot) összehasonlítani csak a ***strcmp()*** függvénnyel (*string compare*) lehet. Ez két sztringparaméterrel rendelkezik, és akkor egyezik a két szöveg, ha a függvény zérus értékkel tér vissza. A példában keressük, hogy az *ut.varos* adattag értéke megegyezike az "*Esztergom*" sztringkonstanssal:

```
if (strcmp(ut.varos, "Esztergom") == 0)
```

A sztringnek értéket csak az ***strcpy()*** függvénnyel adhatunk (*string copy*):

```
strcpy(kocsitip,ut.kocsitipus);
```

A sztring hosszát az ***strlen()*** függvénnyel kérdezhetjük le:

```
db = strlen(kocsitip);
```

A sztringkezelő függvények használatához a *STRING.H* fejállományt kell a programba beszerkeszteni.

```
#include <stdio.h>
#include <string.h>
typedef struct u {
                   char kocsitipus[12];
                   int km;
                   char varos[12];
                 }Uticel;
int main()
{ Uticel ut, LegT, LegK;
  char kocsitip[13];
  int ossz_km = 0;
  FILE *inp;
  kocsitip[0] = ' ';
  inp = fopen("Utvonal.txt","r");
  if (inp == NULL)
  {
    fprintf(stderr,"Utvonal.txt nem letezik!\n"); getchar();
  }
```

```
  else
  {
    printf("100 es 200 km utvonalat megtett autok:\n");
    while (fscanf(inp,"%12s%4d  %12s\n",
                  ut.kocsitipus,&ut.km,ut.varos)!= EOF)
    {
       ossz_km += ut.km;
       if (ut.km > LegT.km) LegT = ut;
       if (ut.km < LegK.km) LegK = ut;
       if (ut.km >= 100 && ut.km <=200)
         printf("%-13s  %5d  %-13s\n",
           ut.kocsitipus,ut.km,ut.varos);
       if (strcmp(ut.varos, "Esztergom") == 0)
          strcpy(kocsitip,ut.kocsitipus);
    }
    fclose(inp);
    printf("\nAz autok altal megtett teljes ut [km]: %d\n",ossz_km);
    printf("\nLegtavolabbi utvonal:\n");

    printf("%-13s  %5d  %-13s\n",
           LegT.kocsitipus,LegT.km,LegT.varos);
    printf("\nLegkozelebbi utvonal:\n");
    printf("%-13s  %5d  %-13s\n",
           LegK.kocsitipus,LegK.km,LegK.varos);
    if (kocsitip[0] != ' ')
      printf("\nEsztergom uticelu auto tipusa: %s\n",kocsitip);
    getchar();
  }
  return 0;
}
```

A program futásának eredménye:

```
100 es 200 km utvonalat megtett autok:
Mercedes          106  Siofok
Suzuki            179  Miskolcs

Az autok altal megtett teljes ut [km]: 793

Legtavolabbi utvonal:
Ford              245  Nyiregyhaza

Legkozelebbi utvonal:
Opel               46  Esztergom

Esztergom uticelu auto tipusa: Opel
```

Írjunk programot, amely a *SZOVEG.TXT* állományból karakterenként olvas és kijelzi, hogy hányadik pozícióban olvasott csillagot és mennyi volt a csillagok száma. A csillagok pozícióját és a darabszámot pedig írjuk ki a *KODOLT.TXT* állományba! A program listázza ki a *KODOLT.TXT* állomány tartalmát! (*KodolFile*)

A *SZOVEG.TXT* szöveges állomány tartalma:

```
az*ido**nem volt***jo****de**azert*jot***kirandultunk.
```

```
#include <stdio.h>
int main()
{
  FILE *inp, *out;
  int hely = 0, csillag = 0;
  char c = ' ';
  inp =fopen("szoveg.txt","r");
  if (inp == NULL)
  {
    fprintf(stderr,"A szoveg.txt allomany nem letezik!");
    getchar();
  }
  else
  {
    out = fopen("kodolt.txt","w");


    while ((c = (char)fgetc(inp)) != '.')
    {
      hely++;
      if ( c == '*') csillag++;

      else
      {
        if (csillag)
        {
          fprintf(out,"%6d%6d\n",hely-csillag, csillag);
          csillag = 0;
        }
      }
    }
    fclose(inp);
    fclose(out);
    printf("kodolt.txt tartalma\n");
    inp = fopen("kodolt.txt","r");
    while( fscanf(inp,"%6d%6d\n",&hely,&csillag)!= EOF)
    {
       printf("%6d%6d\n",hely,csillag);
    }
    fclose(inp);
  }
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
kodolt.txt tartalma
     3     1
     7     2
    17     3
    22     4
    28     2
    35     1
    39     3
```

Írjunk programot, amely a *KOD.TXT* szöveges állományból egész típusú számokat olvas, amely meghatározza, hogy a *SZOVEG.TXT* szöveges állományból hányadik karaktert kell az *UZENET.TXT* állományba kiírni! Használjunk **while** ciklust! A kiírt karaktertől újra kell a számlálást kezdeni! (*Uzenet1*)

A KOD.TXT állomány tartalma:

```
2 3 2 1 2 3 1 3 1 2 2 2 2
```

A SZOVEG.TXT állomány tartalma:

```
*mwea% p+o**tz&^h yv%a#n$
```

```
#include <stdio.h>
int main()
{ FILE *fkod,*fszoveg,*fuzenet;
  int szam;
  char c;
  fkod=fopen("kod.txt","r");
  fszoveg=fopen("szoveg.txt","r");
  fuzenet=fopen("uzenet.txt","w");
  if (fkod != NULL && fszoveg != NULL)
  {
     while (fscanf(fkod,"%d",&szam)!= EOF)
    {
      while (szam--) c=fgetc(fszoveg);
        fputc(c,fuzenet);
    }
    fclose(fszoveg); fclose(fkod); fclose(fuzenet);
    fuzenet=fopen("uzenet.txt","r");
    printf("uzenet.txt allomany tartalma: \n");
    while ((c = (char)fgetc(fuzenet))!=EOF)
    { printf("%c",c); }
    printf("\n");
  }else printf("Hianyos az adatallomany megadasa!\n");
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
uzenet.txt allomany tartalma:
ma potzh van
```

Dolgozzuk át az *UZENET1* programot, hogy a *SZOVEG.TXT* szöveges állományból **for** ciklus használatával olvasson! (*Uzenet2*)

A feladat **for** ciklussal megoldva:

```
    while (fscanf(fkod,"%d",&szam) != EOF)
    { for (i=0; i<szam; i++)
      { c=fgetc(fszoveg); }
      fputc(c,fuzenet);
    }
```

A *SPACE.TXT* szöveges állományt olvassuk be karakterenként, tároljuk egy karaktertömbben, számláljuk meg a karakterek és a duplaszóközök számát! Az eredményeket jelenítsük meg! (*KarOlvas*)

A *SPACE.TXT* állomány tartalma:

```
Ha  jo lesz  az  ido kirandulni  megyunk.
```

```
#include <stdio.h>
int main()
{ int van = 0;
  FILE *inp;
  int c, i = 0, sp2 = 0;
  char s[256];
  inp = fopen("space.txt","r");
  if (inp == NULL)
  {
    fprintf(stderr,"space.txt allomany nem letezik\n"); getchar();
  }
  else
  {
    while((c = fgetc(inp)) != EOF)
    { if (van == 1 && c != ' ') van = 0;
      if (c == ' ') van++;
      if (van == 2) { sp2++, van = 0; }
      s[i] = c;
      i++;
    }
    s[i] = '\0';
    fclose(inp);
    printf("Szoveg: %s\n\nA karakterek szama: %d\n",s, i-1);
    printf("Ket szokoz szama: %d\n",sp2);
  }
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
Szoveg: Ha  jo lesz  az  ido kirandulni  megyunk.

A karakterek szama: 40
Ket szokoz szama: 4
```

11. Bináris állomány I/O műveletei

A bináris állományra vonatkozó rövid elméleti összefoglaló után a bináris fájlokba való írásra és olvasásra mutatunk be különböző feladatokat.

- **Bináris állományok**
  A bináris állományokat bájtonként vagy blokkonként egyaránt kezelhetjük.

  Az állomány megnyitása

  A szöveges állományokhoz hasonlóan a háttértáron lévő fájl tartalmához csak a megnyitás után férhetünk hozzá. A megnyitás hozza létre a kapcsolatot a fájlmutató és az operációs rendszer állománya között. A megnyitáskor definiálódik az állomány típusa és a hozzáférés módja:

```
FILE *fopen(const char *fajlnev, const char *mod);
```

A *fajlnev* paraméter helyén adjuk meg az állomány operációs rendszerben használt nevét, melyben a teljes elérési útvonal is szerepelhet.
A *mod* paraméter szintén sztring, amely a fájl elérését és típusát meghatározza.

Bináris	*A **mod** paraméter jelentése*
`"rb"`	Létező állomány megnyitása.
`"wb"`	Új állomány megnyitása. Ha az állomány már létezik, akkor a tartalma elvész.
`"ar"`	Az állomány megnyitása a végéhez való hozzáírásra, tehát az állomány bővítése. Ha a fájl nem létezik, akkor megnyitáskor létrejön.
`"r+b"`	A létező állomány megnyitása írásra és olvasásra (*update*).
`"w+r"`	Új fájl megnyitása írásra és olvasásra (*update*). Ha az állomány már létezik, akkor a tartalma elvész.
`"a+r"`	Az állomány megnyitása a végén végzett írásra és olvasásra (*update*). Ha a fájl nem létezik, akkor most létrejön.

Ha az olvasni kívánt állomány nem létezik, akkor hibajelzést adunk, például:

```
if ((f = fopen("Utazas.dat","rb")) == NULL)
{
  fprintf(stderr,"Utazas.dat allomany nem letezik!\n");
   . . .
}
```

Bináris módú adatfolyam esetén nincs '\n' konverzió. Tehát a '\n' karakter kiírásakor mindig 13 kódú karakter íródik ki. Az operációs rendszerben a '\n' megfelelő kódsorozat is konvertálás nélkül kerül olvasásra. Tehát például a Windows rend-

szerekben bináris adatfolyamban érkező CR LF sorozat (13 10) a '\n'-re történő konverzió (1 karakter) helyett változatlan formában kerül beolvasásra.

Memóriablokk- írása és olvasása

A memóriablokkok állományba való írása és állományból történő visszaolvasása az *fwrite()*, illetve az *fread()* függvények használhatók, melyek prototípusa:

```
size_t fwrite(const void *ptr, size_t meret, size_t n,
              FILE* adatfolyam);

size_t fread(const void *ptr, size_t meret, size_t n,
             FILE* adatfolyam);
```

Mindkét függvény adott *meret* méretű adatelemekkel dolgozik, amelyekből *n* darabot ír vagy olvas a hívás során. Az adatokat tartalmazó memóriaterület kezdetét a *ptr* mutató jelöli ki.

A függvények visszatérési értéke a kiírt, illetve a beolvasott elemek számát (nem bájtok) adja meg. Ha ez az érték kisebb, mint az általunk kijelölt (*n*), akkor fájlvége, vagy fájlhiba eseményre lehet következtetni.

Pozicionálás az állományban

Az állomány elejére a *rewind()* függvénnyel pozicionálunk:

```
rewind(f);
```

Tetszőleges pozíciót választhatunk ki az *fseek()* függvénnyel:

```
int fseek(FILE *adatfolyam, long tavolsag, int honnan);
```

A *tavolsag* az állománypozíció relatív távolságát tartalmazza, a *honnan* paraméter által kijelölt pozícióhoz képest. A *honnan* paraméter lehetséges értékei:

SEEK_CUR – az aktuális pozícióhoz képest,
SEEK_END – a fájl végéhez képest,
SEEK_SET – a fájl elejéhez képest.

Az aktuális pozíciót lekérdezhetjük az *ftell()* függvénnyel:

```
long ftell(FILE *adatfolyam);
```

11.1 Bájtonkénti feldolgozás

Olvassunk be ponttal végződő szöveget karakterenként és írjuk ki bináris fájlba! Olvassuk vissza a fájlból és számláljuk meg a szövegben található magánhangzókat, mássalhangzókat és egyéb jeleket (;:,.) fűzzük! (*KarFile*)

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <ctype.h>
#include <string.h>

void Ir(char FileN[]);
void Olvas(char FileN[], int *e, int *mgh, int *msh, int *jk);
int main()
{
 char FileName[13];
 char c;
 int  egyeb,maganh,massalh,jelek;
 printf("Allomany neve: "); scanf("%s",FileName);
 Ir(FileName);
 Olvas(FileName,&egyeb,&maganh,&massalh,&jelek);
 printf("\n\nStatisztika\n");
 printf("Maganhangzok szama : %d\n",maganh);
 printf("Massalhangzok szama: %d\n",massalh);
 printf("Jelek (;:,.)       : %d\n",jelek);
 printf("Egyeb jelek, szokoz: %d\n",egyeb);
 fflush(stdin); getchar();
 return 0;
}
void Ir(char FileN[])
{
 FILE *f;
 char ch;
  f= fopen(FileN,"wb");
  printf("A szoveg veget a . jelzi ");
  ch = '*';
  printf("\n:");
  while (ch != '.')
  {
    ch = getc(stdin);
    fputc(ch,f);
  }
  fclose(f);
}
void Olvas(char FileN[], int *e, int *mgh, int *msh, int *jk)
{
 FILE *f;
 char ch;
 if ((f= fopen(FileN,"rb")) == NULL)
 {
   fprintf(stderr,"A %s fajl nem letezik!",FileN);
   exit(-1);
 }
 *e = 0; *mgh = 0; *msh = 0; *jk = 0;
 printf("\nA fajlbol olvasva:");
 while (!feof(f))
 {
   ch = fgetc(f);
   printf("%c",ch);
   if (isalpha(ch))
   {
     if (strchr("aoiueAOIUE",ch))
      (*mgh)++;
```

```
    else
       (*msh)++;
    }
    else if (isspace(ch))
    {
      (*e)++;
    }
    else if (ch == ';' || ch == ':' ||
             ch == '.' || ch == ',')
      (*jk)++;
  }
  fclose(f);
}
```

A program futásának eredménye:

```
Allomany neve: mondat.dat
A szoveg veget a . jelzi
:Holnap kirandulni megyunk, ha nem esik az eso.

A fajlbol olvasva:
Holnap kirandulni megyunk, ha nem esik az eso.

Statisztika
Maganhangzok szama : 15
Massalhangzok szama: 22
Jelek (;:,.)       : 2
Egyeb jelek, szokoz: 8
```

11.2 Numerikus értékek bináris fájlba való tárolása

Olvassunk be billentyűzetről max. 100 darab egész számot és írjuk ki a megadott nevű állományba binárisan. Az állományból az adatokat töltsük vissza és számláljuk meg a pozitív, negatív és a zérus adatok számát! (*NumFile*)

```
#include <stdio.h>
#include <stdlib.h>
typedef int tomb[100];
void Ir(char FileN[]);
int Olvas(char FileN[],tomb t, int *poz,
          int *neg, int* zero, int * n);
int main()
{
 char FileName[13];
 tomb x;
 int  db,p,z,m;
 printf("Allomany neve: "); scanf("%s",FileName);
 Ir(FileName);
 if (!Olvas(FileName,x,&p,&m,&z,&db))
 {
   printf("negativ elemek szama: %d\n",m);
   printf("pozitiv elemek szama: %d\n",p);
```

```
    printf("zerus elemek szama  : %d\n",z);
    fflush(stdin); getchar();
    return 0;
  }
 return 1;
}
void Ir(char FileN[])
{
  FILE *f;
  int i,k,n;
  f= fopen(FileN,"wb");
  printf("Elemek szama: "); scanf("%d",&n);
  for(i = 0 ; i<n; i++)
  {
    printf("adat: "); scanf("%d",&k);
    fwrite(&k,sizeof(k),1,f);
  }
  fclose(f);
}
int Olvas(char FileN[],tomb t, int *poz,
          int *neg, int* zero,int * n)
{
  FILE *f;
  int k;
  if ((f= fopen(FileN,"rb")) == NULL)
  {
    fprintf(stderr,"A %s fajl nem létezik!",FileN);
    getchar(); return 1;
  }
  *n = 0;*poz = 0; *neg = 0; *zero = 0;
  while ( fread(&k, sizeof(k), 1, f)>0 )
  {
    (*n)++;
    t[*n] = k;
    if (t[*n] < 0) (*neg)++;
    if (t[*n] > 0) (*poz)++;
    if (t[*n] == 0) (*zero)++;
  }
  fclose(f);
  return 0;
}
```

A program futásának eredménye:

```
Allomany neve: adat.dat
Elemek szama: 5
adat: -2
adat: 3
adat: 0
adat: 5
adat: -4
negativ elemek szama: 2
pozitiv elemek szama: 2
zerus elemek szama  : 1
```

11.3 Struktúrák bináris állományba történő írása és olvasása

A *Lista* feladatot dolgozzuk át úgy, hogy a *Henger* struktúrák adatait először egy bináris állománya írjuk, majd visszaolvasva listába fűzzük! (*ListaFile*)

```
typedef struct hr {
                    int sugar;
                    int magassag;
                    double terfogat;
                    struct hr *kov, *elozo;
                   }Henger;

/*LISTAFG.H*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define PI 3.141592654
typedef struct hr {
                    int sugar;
                    int magassag;
                    double terfogat;
                    struct hr *kov, *elozo;
                   }Henger;

Henger* OlvasFileHenger(Henger **u);
int IrFileHenger(void);
int MentesAllomanyba(Henger *e);
void KiirElore(Henger* e);
void KiirHatra(Henger* u);
void KeresMinMax(Henger *e);
void BeszurV(Henger **e, Henger **v, int r, int h);
Henger* Torol(Henger *e, Henger **u, int x1, int y1);

/*LISTAFG.C*/
#include "Listafg.h"
#include <process.h>  /* az exit() használatához */
```

Az *IrFileHenger()* függvény a *HENGEREK.DAT* bináris állományba írja a beolvasott *Henger* struktúrákat.

```
int IrFileHenger(void)
{
 int i, db;
 Henger uj;
 FILE *fh;
 fh = fopen("Hengerek.dat","wb");
 if (fh == NULL)
 {
   fprintf(stderr,"Allomany megnyitasi hiba\n");
   getchar();
   return 1;
 }
```

```
 else
 {
   printf("Adatok szama: ");
   scanf("%d",&db);
   for (i = 0; i< db; i++)
   {
     printf("\nsugara: "); scanf("%d",&uj.sugar);
     printf("magassaga: "); scanf("%d",&uj.magassag);
     uj.terfogat = pow(uj.sugar,2)*PI*uj.magassag;
     fwrite(&uj,sizeof(Henger),1,fh);
   }
   fclose(fh);
  }
 return 0;
}
```

Az *OlvasFileHenger()* függvény a *HENGEREK.DAT* bináris fájlból beolvassa a *Henger* struktúrákat és láncba fűzi. A lánc elejére mutató pointert függvényértékként, a lánc utolsó elemére mutató pointert pedig a paraméteren keresztül adja vissza.

```
Henger* OlvasFileHenger(Henger **u)
{FILE *fh;
 Henger h, *uj, *elso = NULL, *utolso = NULL;
 fh = fopen("Hengerek.dat","rb");
 if (fh == NULL)
 {
   fprintf(stderr,"Allomany nem letezik\n");
   getchar();
 }
 else
 {
   while (fread(&h,sizeof(Henger),1,fh) == 1)
   {
     uj = (Henger*)malloc(sizeof(Henger));
     if (uj == NULL)
     {
       printf("Memoriafoglalas nem sikerult!\n");
       getchar(); exit(1);  /* kilépés a programból */
     }
     uj->sugar = h.sugar;
     uj->magassag = h.magassag;
     uj->terfogat = pow(uj->sugar,2)*PI*uj->magassag;
     uj->kov = NULL; uj->elozo = NULL;
     if (elso == NULL) elso = uj;
     else
     {
       utolso->kov = uj; uj->elozo = utolso;
     }
     utolso = uj;
   }
   *u = utolso;
   fclose(fh);
 }
 return elso;
}
```

A *MentesAllomanyba()* függvény a módosított lánc elemeivel felülírja a *HENGEREK.DAT* bináris állományt.

```
int MentesAllomanyba(Henger *e)
{
 int i, db;
 FILE *fh;
 fh = fopen("Hengerek.dat","wb");
 if (fh == NULL)
 {
   fprintf(stderr,"Allomany nem letezik'\n");
   getchar();
   return 1;
 }
 else
 {
  while(e != NULL)
  {
    fwrite(&e,sizeof(Henger),1,fh);
    e = e->kov;
  }
  fclose(fh);
 }
 return 0;
}
```

A *KeresMinMax()* függvény megkeresi a legnagyobb és a legkisebb térfogatú hengert, és megjeleníti az adataival együtt.

```
void KeresMinMax(Henger *e)
{
  Henger *LegK, *LegN;
  if (e != NULL)
  {
    LegK = e;
    LegN = e;
    while (e != NULL)
    {
      if (e->terfogat > LegN->terfogat) LegN = e;
      if (e->terfogat < LegK->terfogat) LegK = e;
      e = e->kov;
    }
    printf("\nLegnagyobb terfogatu henger adatai:\n");
    printf("sugar: %d  magassag: %d  terfogat: %8.2lf\n",
           LegN->sugar, LegN->magassag, LegN->terfogat);
    printf("\nLegkisebb terfogatu henger adatai:\n");
    printf("sugar: %d  magassag: %d  terfogat: %8.2lf\n",
           LegK->sugar, LegK->magassag, LegK->terfogat);
  }else printf("\nA lista ures\n");
}
```

A *KiirElore()* függvény a láncolt lista elemeit az első elemtől kezdve előre haladva jeleníti meg.

```
void KiirElore(Henger* e)
{
  if (e != NULL)
  {
    printf("\nA lancolt lista elemei (elore)\n");
    printf("   sugar  magassag  terfogat    \n");
    while (e != NULL)
    {
      printf("%8d  %8d  %8.2lf\n",e->sugar,e->magassag,e->terfogat);
      e = e->kov;
    }
  }else printf("\nA lista ures\n");
}
```

A *KiirHatra()* függvény a láncolt lista elemeit az utolsó elemtől kezdve visszafelé lépkedve jeleníti meg.

```
void KiirHatra(Henger* u)
{ if (u != NULL)
  {
    printf("\nA lancolt lista elemei (visszafele)\n");
    printf("   sugar  magassag\n");
    while (u != NULL)
    {
      printf("%8d  %8d\n",u->sugar,u->magassag);
      u = u->elozo;
    }
  }else printf("\nA lista ures\n");
}
```

A *BeszurV()* függvény a beszúrandó henger adatait megadva (*r, h*) az első elem mögé szúrja be. A paraméterlistán megadott *e* és *v* paramétereken keresztül a lánc elejére, illetve a végére mutató, módosított mutatókat adja vissza a függvény. A függvény üres lánc esetén is működik.

```
void BeszurV(Henger **e, Henger **v, int r, int h)
{  Henger *uj;
   Henger *elso, *utolso;
   elso = *e;
   utolso = *v;
   uj = (Henger*)malloc(sizeof(Henger));
   if (uj == NULL)
     {
       printf("Memoriafoglalas nem sikerult!\n");
       fflush(stdin); getchar(); exit(1);
     }
   uj->sugar = r;
   uj->magassag = h;
   uj->terfogat = pow(uj->sugar,2)*PI*uj->magassag;
   uj->kov = NULL;
   uj->elozo = NULL;
```

```
    if (elso == NULL && utolso == NULL )
    {
       printf("Ures a lanc, az allomany meg nincs betoltve\n");
       getchar();
    }
    else
    {
      elso->elozo = uj;
      uj->kov = elso;
      *e = uj;
      *v = utolso;
    }
}
```

A *Torol()* függvény a törlendő henger adatait megkapva (*r, h*) törli a hengert a láncból. Ha nem találja, figyelmeztető jelzést ad.

```
Henger* Torol(Henger *elso, Henger **utolso, int r, int h)
{
  Henger *p, *pkov;
  int talalt = 0;
  p = elso;
  while (p!= NULL)
  {
    if (p->sugar == r && p->magassag == h)
    {
      talalt++;
      if (p->elozo == NULL && p->kov == NULL)
      {
        elso = NULL; *utolso = NULL; free(p); break;
      }
      if (p->elozo != NULL) (p->elozo)->kov = p->kov;
      else elso = p->kov;
      if (p->kov != NULL) (p->kov)->elozo = p->elozo;
      else *utolso = p->elozo;
      free(p); break;
    }
    p = p->kov;
  }
  if (talalt == 0)
    printf("\nsugar: %d magassag: %d nincs a listaban'\n",r,h);
  return elso;
}

/*MAIN.C*/
#include "Listafg.h"
int main()
{
  int i, r, h, tdb;
  Henger *pelso = NULL, *putolso = NULL, *p, *p1;

  if (!IrFileHenger())
  {
    printf("Az allomany betolve!\n");
    pelso = OlvasFileHenger(&putolso);
```

```
    KiirElore(pelso);
    KeresMinMax(pelso);
  }
  else
  {
    printf("Az allomany irasa nem sikerult!\n");
  }
  p = pelso;
  while (p!= NULL)
  {
    p1 = p->kov;
    free(p);
    p = p1;
  }
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
Adatok szama: 3

sugara: 5
magassaga: 6

sugara: 1
magassaga: 1

sugara: 2
magassaga: 3
Az allomany betolve!

A lancolt lista elemei (elore)
   sugar  magassag  terfogat
       5         6    471.24
       1         1      3.14
       2         3     37.70
Legnagyobb terfogatu henger adatai:
sugar: 5  magassag: 6  terfogat:   471.24

Legkisebb terfogatu henger adatai:
 sugar: 1  magassag: 1  terfogat:     3.14
```

12. Numerikus módszerek

A mindennapi életben több műszaki, közgazdasági és természettudományi feladat visszavezethető egy-egy matematikai módszerre, például függvények integrálására, függvények ábrázolásához szükséges interpolációra, nemlineáris egyenletek, illetve lineáris egyenletrendszerek megoldására. Az ilyen feladatok számítási módszereivel a numerikus analízis foglalkozik.

Ebben a alfejezetben néhány numerikus módszer matematikai hátterét és C nyelven történő megvalósítását ismertetjük. A programrészletek és egy-egy feladat futtatási eredménye az elmélet mélyebb megértését segíti.

12.1 Nemlineáris egyenlet numerikus megoldása

A feladat az `f(x) = 0` tetszőleges, nemlineáris egyenlet gyökeinek meghatározása. A gyakorlatban mindig teszünk kikötéseket f(x)-re és a keresett gyök (gyökök) elhelyezkedésére vonatkozóan.

Newton-Raphson-módszer és a húrmódszer együttes alkalmazása

A *Newton-Raphson*-módszer gyorsasága ötvözhető a húrmódszer garantált konvergenciájával. Egy kezdeti $[x_0,x_1]$ intervallumból kiindulva, amely a gyököt közrefogja, tehát $f(x_0)*f(x_1) < 0$, *Newton-Raphson*-iterációt végzünk:

$$x_2 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Ha x_2 kiesik az $[x_0, x_1]$ intervallumból, akkor elvetjük, és húrmódszerrel számítunk az intervallumba eső új x_2 értéket. Ezután az intervallumot úgy szűkítjük, hogy a gyököt továbbra is közrefogja. A leállási feltétel:

$$|f(x_2)| \leq \varepsilon$$

A módszert használó függvény:

```
int NewtonRH(double xe, double xv, double eps, double *gyok,
             fp f1, fp df1)
```

A függvény paraméterei:

xe	–	az intervallum eleje,
xv	–	az intervallum vége,
**gyok*	–	a nemlineáris egyenlet gyöke, a függvény eredménye,
eps	–	leállási feltétel, a gyök pontossága,

f1 – a nemlináris egyenlet függvényének címe,
df1 – a nemlineáris egyenlet derivált-függvényének címe.

A függvény visszatérési értékének jelentése:

0 – ha hiba történt,
1 – ha az egyenletnek megtalálta a gyökét.

Határozzuk meg az

$$y = x^3 + x - 0.7$$

nemlineáris egyenlet gyökét a [0,1] intervallumban 10^{-6} hibakorláttal!

Az *fx()* függvény tartalmazza a nemlineáris egyenletet, a *dfx()* függvényben pedig a nemlineáris egyenlet deriváltját kell megadni.

A **typedef** utasítással definiáljuk a függvényre mutató pointer típusát (**fp*):

```
typedef double f(double x);
typedef f *fp;
```

```
/*NEMLIN*/
#include <stdio.h>
#include <math.h>

typedef double f(double x);
typedef f *fp;

double fx(double x);
double dfx(double x);
int NewtonRH(double xe,double xv, double eps, double *gyok,
             fp f1, fp df1);
int main()
{ double xk, xv, eps2 = 1e-6, gyok;
  printf("Gyok meghatarozasa Newton-Raphson- es a hurmodszer\n");
  printf("egyuttes alkalmazasa\n");
  printf("\nIntervallum eleje: "); scanf("%lf",&xk);
  printf("Intervallum vege: "); scanf("%lf",&xv);
  if (NewtonRH(xk,xv,eps2,&gyok,fx,dfx))
  {
    printf("\nGyok: %lf\n",gyok);
  }
  fflush(stdin); getchar();
  return 0;
}

/* nemlineáris egyenlet */
double fx(double x)
{
  return pow(x,3)+x-0.7;
}
```

```
/* nemlineáris egyenlet deriváltja */
double dfx(double x)
{
  return 3*pow(x,2)+1;
}

int NewtonRH(double xe, double xv, double eps, double *gyok,
             fp f1, fp df1)
{
  double x0, x1, y0, y1, y2;
  x0 = xe;
  x1 = xv;
  do
  {
    y0 = (*f1)(x0); /* azonos az f1(x0) hivással */
    y1 = (*f1)(x1);
    y2 = (*df1)(x0);
    if (fabs(y1 - y0) < 1e-6)
    {
       printf("Nincs gyok!\n"); getch();
       return 0;
    }
    if (fabs(y2) < 1e-6)
    {
       printf("Nincs gyok!\n"); getch();
       return 0;
    }
    x1 -= y1 * (x1 -x0)/(y1 - y0);
    x0 -= y0/y2;
  }while (eps < fabs( x1 - x0));
  *gyok = x1;
  return 1;
}
```

A program futásának eredménye:

```
Gyok meghatarozasa Newton-Raphson- es a hurmodszer
egyuttes alkalmazasa

Intervallum eleje: 0
Intervallum vege: 1

Gyok: 0.541351
```

12.2 Numerikus integrálás (numerikus kvadratúra)

A numerikus integrálás feladata, a

$$\int_a^b f(x)\,dx$$

határozott integrál kiszámítása. Akkor alkalmazzuk, ha

- a függvény pontokban ismert (mérési eredmények),
- a függvénynek nem létezik primitív függvénye, például:

$$\sqrt{1+x^3};\quad e^{x^2};\quad \frac{1}{\ln(x)};\quad \frac{\sin(x)}{x},$$

- a bonyolult függvényt a Lagrange-polinomjával közelítjük.

A numerikus integrálás formulája vagy más elnevezéssel a numerikus kvadratúra formulája:

$$\int_a^b y(x)dx = \sum_{i=0}^{n} c_i \cdot y(x_i) + E(y)$$

A hiba

$$E(y) = \int_a^b y(x)dx - \sum_{i=0}^{n} c_i \cdot y(x_i)$$

A numerikus integrálás egyik célja az E(y) hiba csökkentése.

Integrálási módszernek nevezzük a c_i súlyok és az x_i alappontok bizonyos megválasztási rendszerét. Integrálási módszer pontossági foka vagy a pontossági rendje n, ha a módszerrel legfeljebb n-edfokú polinomot integrálva teljesül az

$$E(y) = 0\,.$$

Az E(y) csökkentésére két lehetőségünk van:

- magasabb fokszámú közelítést alkalmazunk.
- részintervallumokra bontjuk az integrált.

A numerikus integrálási módszereket egyszerű és összetett kvadratúramódszerek csoportjába oszthatjuk. Az egyszerű kvadratúramódszerben az integrálás intervallumát egyetlen intervallumnak tételezzük fel, és az integrálási módot határozza meg az x_i-k száma.

Az összetett integrálási módszerek esetében az integrálás intervallumát részintervallumokra osztjuk, és ezekre a részintervallumokra alkalmazzuk az egyszerű kvadratúramódszereket. A teljes integrál a részintegrálok összege.

Mind az egyszerű, mind az összetett kvadratúramódszerek esetén a célunk az E hiba csökkentése. Ezt kétféleképpen érhetjük el. Magasabb pontossági rendű kvadratúramódszert választunk, vagy az integrálás intervallumát részintervallumokra osztjuk, azaz összetett kvadratúramódszert alkalmazunk. Mindkét esetben az egész integrálási intervallumra eső x_i -k száma megnő. A számítási időt elsősorban az x_i-k száma határozza meg.

Kérdés az, hogy ugyanakkora számítási idő mellett (ugyanannyi x_i) az egyszerű vagy az összetett kvadratúra a pontosabb. A részintervallumok számának növelésével majdnem minden függvényosztályra az E hiba gyorsan csökken, ugyanakkor az integrálási módszer nem válik bonyolulttá. Ezért a gyakorlatban majdnem mindig az összetett kvadratúra-módszert alkalmazunk. Bizonyítható, hogy azonos x_i szám mellett az összetett kvadratúra nemcsak egyszerűbb, de pontosabb is.

A numerikus integrálási módszerek osztályozásának egy másik szempontja, hogy az x_i alappontok elhelyezkedése:
- a felhasználó által választott, például ekvidisztáns (egyenletes), vagy
- a módszer szempontjából optimális.

Táblázatos $y(x)$ megadás esetén célszerű az adott, legtöbbször ekvidisztáns osztást használni. Az ekvidisztáns lépésközzel egyszerűbb a számítás. Az x_i-k optimális elhelyezése elvileg nagyobb pontosságot biztosít, a gyakorlatban azonban mégis többször használják az ekvidisztáns lépésközt, mivel a pontok legtöbbször adottak.

Egyenletes lépésközzel működik a Newton-Cotes-módszer, speciális lépésközt használnak a Gauss, Gauss-Csebisev, Gauss-Laguerre és a Gauss-Hermite módszerek.

Newton-Cotes kvadratúramódszerek

A Newton-Cotes féle kvadratúramódszernél a függvényt egy n-ed fokú Lagrange polinommal helyettesítjük. Az integrálásnál az $y(x)$ függvényt az $l_i(x) \cdot y(x_i)$ Lagrange-féle interpolációs polinommal közelítjük. Mivel az integrálás és az összegzés felcserélhető, valamint az $y(x_i)$-k konstansok, ezért

$$\int_a^b y(x)dx$$

az integrál a következő alakra hozható:

$$\int_a^b \sum_{i=0}^{n} l_i(x) \cdot y(x_i)dx = \sum_{i=0}^{n} y(x_i) \cdot \int_a^b l_i(x)\,dx$$

ahol

$$l_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}$$

Alkalmazva a

$$c_i = \int_a^b l_i(x)\,dx \text{ helyettesítést}$$

$$\int_a^b y(x)dx = \sum_{i=0}^{n} c_i \cdot y(x_i) \qquad (1)$$

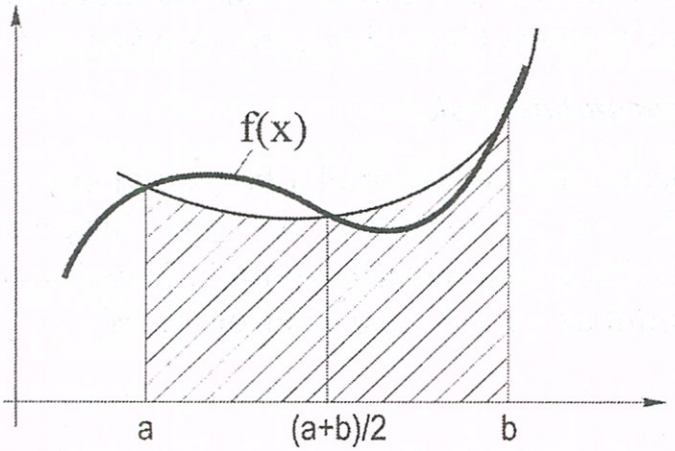
(Az $l_i(x)$-k integrálása kis *n*-ek esetén egyszerű.)

A *MINTA12_1* mintafeladat a kis- és a nagy *Simpson*-formulák a használatát mutatja be.

Egyszerű (kis) Simpson-formula

A görbe alatti területet másodfokú polinomokkal (parabola) határolt trapézszerű alakzattal helyettesítjük.

A *Newton-Cotes*-formulából n = 2 választás mellett,



Egyszerű Simpson-formula alkalmazása

az eddigiekhez hasonlóan, bizonyítható (illetve a Lagrange-féle interpolációs polinomokból integrálással levezethető) a másodfokú (parabola) közelítést használó kvadratúraképlet, amelyet az egyszerű vagy kis Simpson-formula néven ismerünk:

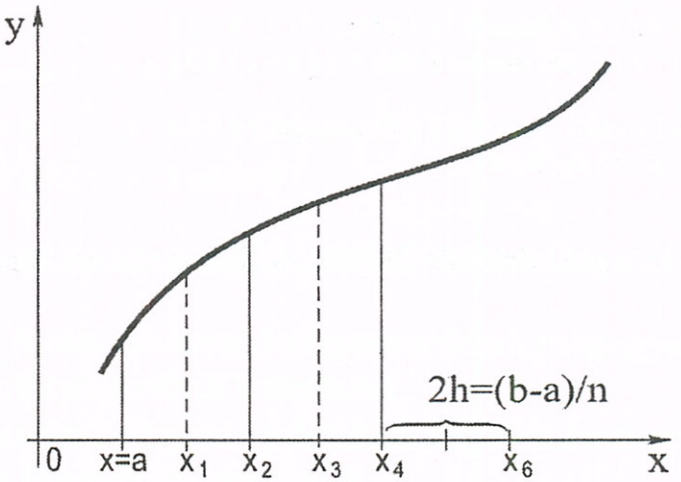
$$\int_a^b y(x)\,dx \approx \frac{b-a}{6}\left[y(a) + 4 \cdot y\left(\frac{a+b}{2}\right) + y(b)\right]$$

A módszert használó programrészlet:

```
void KisSimpson(double alsoh, double felsoh, double *Eredmeny, fp f1)
{
  *Eredmeny = (felsoh - alsoh)/6.*((*f1)(alsoh) + 4.*(*f1)((alsoh
              + felsoh)/2.) + (*f1)(felsoh));
}
```

Összetett (nagy) Simpson-formula

Az összetett Simpson-formulát akkor kapjuk, ha *n* számú, *2h* hosszúságú intervallumszakaszra képezzük az összeget.



Összetett Simpson formula alkalmazása

$$\int_a^b y(x)\,dx \approx \frac{2\cdot h}{6}\left[y(x_0)+4y(x_1)+y(x_2)+y(x_2)+4y(x_3)+y(x_4)\ldots\right]=$$

$$=\frac{h}{3}\left[y(x_0)+4y(x_1)+2y(x_2)+4y(x_3)+2y(x_4)+\right.$$

$$\left.+\ldots 2y(x_{n-2})+4y(x_{n-1})+y(x_n)\right]$$

Tehát az összetett Simpson-formula esetén:

- az *a* és *b* pontok egyszeres súllyal ($x_0 = a$; $x_n = b$),
- a páratlan sorszámúak 4-szeres súllyal,
- míg, a páros sorszámúak (mivel ezek a megelőző tartomány végei, illetve a következő tartomány első pontjai) kétszeres súllyal rendelkeznek.

A *Simpson*-formula igen nagy előnye, hogy egyszerűsége mellett nagy a pontossága.
A *Simpson*-formula hibája:

$$E(y)=\frac{1}{180}\left(\frac{b-a}{n}\right)^4 y^{IV}_{max}(x)$$

Beláthatjuk, ha a *h* lépéstávolságot kétszer kisebbre vesszük, akkor a *Simpson* képlet hibája kb. 16-odára csökken. Ezért a *h/2* lépéstávolsággal számított integrál értéke egy értékes jeggyel többet tartalmaz, mint a *h* lépéssel számított integrál értéke.

A módszert használó programrészlet:

```
void NagySimpson(double alsoh, double felsoh, int osztas,
                 double *Eredmeny, fp f1)
{
  int i;
  double s = 0, h;
  h = (felsoh - alsoh)/osztas;
  for (i = 1; i< osztas; i++)
  {
    s +=  4.*(*f1)(alsoh + i*h)*(i % 2)
        + 2.*(*f1)(alsoh+i*h)*((i+1)%2);
  }
  s += (*f1)(alsoh) + (*f1)(felsoh);
  *Eredmeny = s * h/3.0;
}
```

Számítsuk ki az alábbi integrál értékét 3000-res felosztás esetén!

$$\int_0^1 \frac{4}{1+x^2}\,dx$$

```
/*INTEGRAL*/
#include <stdio.h>
#include <math.h>

typedef double f(double x);
typedef f *fp;
double fg1(double x);
void KisSimpson(double alsoh, double felsoh,
            double *Eredmeny, fp f1);
void NagySimpson(double alsoh, double felsoh, int osztas,
            double *Eredmeny, fp f1);
int main()
{
  double alsoH, felsoH, ered;
  int felosztas;
  printf("Integral szamitasa \n\n");
  printf("Also hatar : "); scanf("%lf",&alsoH);
  printf("Felso hatar: "); scanf("%lf",&felsoH);
  printf("Intervallum felosztasa: "); scanf("%d",&felosztas);
  KisSimpson(alsoH, felsoH, &ered,fg1);
  printf("\nKis Simpson-modszerrel : %lf\n",ered);
  NagySimpson(alsoH, felsoH, felosztas, &ered,fg1);
  printf("\nNagy Simpson-modszerrel: %lf\n",ered);
  fflush(stdin); getchar();
  return 0;
}
```

```
/*pi értékét számítjuk ki integrállal
 alsó határ: 0
 felső határ: 1
 osztások száma: 3000 */

double fg1(double x)
{
   return 4.0/(1. +x*x);
}
 . . .
```

A program futásának eredménye:

```
Integral szamitasa

Also hatar : 0
Felso hatar: 1
Intervallum felosztasa: 3000

Kis Simpson-modszerrel : 3.133333

Nagy Simpson-modszerrel: 3.141593
```

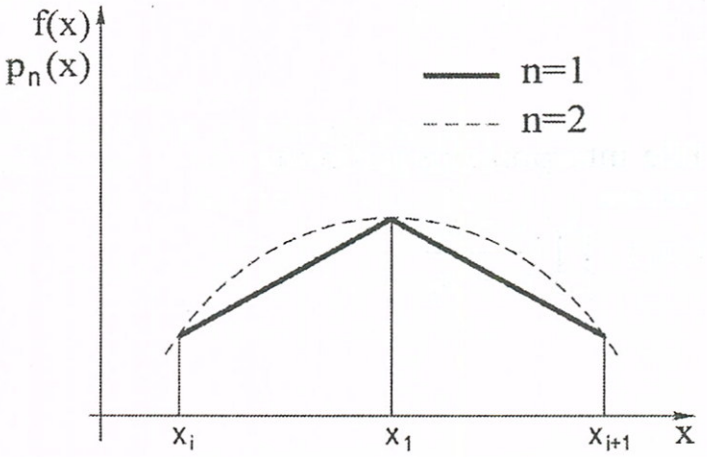
12.3 Interpoláció

Ha a közelítő függvény értéke az ismert alappontokban megegyezik a közelített függvény értékével, akkor interpolációról beszélünk

- Mérési sorozatok esetén a függvény értékeit véges számú pontban ismerjük, a mérési pontok között a függvényt közelítjük.
- A függvények és a mérési eredmények ábrázolásánál véges számú pontot veszünk fel és ezeket kötjük össze.

Lagrange-interpolációs módszer:

A két alappontot (x_0, x_1) felhasználó közelítés ($n=1$) a két pont közötti egyenessel közelít. Belátható, hogy a három alappontos közelítés ($n=2$) egy másodfokú függvény, parabola lesz.



Két alappontot felhasználó közelítés

A négy alappontos közelítés harmadfokú függvény, és így tovább. *n+1* alappont esetén bebizonyítható, hogy a közelítés *n*-edfokú polinom lesz. A polinom együtthatóinak kiszámítására *Lagrange* dolgozott ki eljárást.

Állítsuk elő a polinomot az alábbi alakban:

$$p_n(x) = \sum_{i=0}^{n} l_i(x) \cdot f(x_i)$$

és

$$l_i(x_j) = 0, \text{ ha } i \neq j \qquad l_i(x_j) = 1, \text{ ha } i = j$$

vagyis az l_i függvények a saját alappontban 1, a többi alappontban 0 értéket adnak. Akkor az x_i -edik alappontban:

$$p_n(x_i) = 0 \cdot f(x_0) + 0 \cdot f(x_1) + \ldots + 1 \cdot f(x_i) + \ldots + 0 \cdot f(x_n) = f(x_i)$$

vagyis a feltétel teljesül. A közelítő polinom az alappontokban (valamennyi alappontban) felveszi a közelített függvény értékét, az alappontok között pedig valamilyen közelítést ad. A kérdés már csak az, hogyan lehet az $l_i(x)$ függvényeket előállítani.
Legyen

$$l_i(x) = \frac{(x-x_0)\cdot(x-x_1)\ldots(x-x_{i-1})\cdot(x-x_{i+1})\ldots(x-x_n)}{(x_i-x_0)\cdot(x_i-x_1)\ldots(x_i-x_{i-1})\cdot(x_i-x_{i+1})\ldots(x_i-x_n)}$$

Ekkor az x_0 alappontban a számláló első tényezője nulla, x_1 alappontban a második és így tovább, tehát mivel csak az $(x - x_i)$ tényező hiányzik, ez minden más alappontban nulla lesz. Az $x = x_i$ helyen pedig a számláló és a nevező azonos, $l_i(x_i) = 1$. Az $l_i(x)$ *Lagrange*-féle interpolációs alappolinomokat tömörebben felírva

$$l_i(x) = \prod_{\substack{j=0 \\ i \neq j}}^{n} \frac{x - x_j}{x_i - x_j} \quad i = 0, \ldots n$$

így a *Lagrange*-féle interpolációs polinom:

$$p_n(x) = \sum_{\substack{i=0 \\ i \neq j}}^{n} f(x_i) \cdot \prod_{j=0}^{n} \frac{x - x_j}{x_i - x_j}$$

```
/*LAGRANGE*/
#include <stdio.h>
#include <stdlib.h>
int main()
{ double *x = NULL, *y = NULL, xh, yh;
  int i, n;
  do
   {
     printf("\nElemek szama (min. 2): ");  scanf("%d",&n);
   } while (n < 2);
  x = (double*)malloc(n*sizeof(double));
  y = (double*)malloc(n*sizeof(double));
  if (x == NULL || y == NULL)
  { printf("Memoriafoglalas nem sikerult!\n");
    fflush(stdin); getchar(); return 1;
  }
  for (i = 0; i<n; i++)
  { printf("\nx[%2d] = ",i); scanf("%lf",&x[i]);
    printf("y[%2d] = ",i); scanf("%lf",&y[i]);
  }
  printf("\nIterpolalando ertek (xh): "); scanf("%lf",&xh);
  if (Lagrange(x,y,n,xh,&yh) == 0)
  {
    printf("\nLagrange-interpolacio\n");
    printf("Interpolacio helye(xh): %lf\n",xh);
    printf("Intepolalt ertek (yh) : %lf\n",yh);
  }
  free(x); free(y);
  fflush(stdin); getchar();
  return 0;
}

/*Lagrange interpoláció*/
int Lagrange(double *x, double *y, int n, double xh, double *yh)
{ int i, j;
  double p = 0.0, s;
  for (i = 0; i< n; i++)
  { s = 1.0;
    for (j = 0; j<n; j++)
    { if (i != j)
      { if (x[i] == x[j])
        { printf("Hibajelzes:\n");
          printf("Azonosak az x koordinatak: \n");
          printf("%2d.elem : %lf\n",i,x[i]);
          printf("%2d.elem : %lf\n",i+1,x[i+1]); return 1;
        }
        s *=(xh-x[j])/(x[i] - x[j]);
      }
    }
    p += y[i]*s;
  }
  *yh= p;
  return 0;
}
```

A program futásának eredménye:

```
Elemek szama (min. 2): 4

x[ 0] = 1
y[ 0] = 1

x[ 1] = 2
y[ 1] = 1.8

x[ 2] = 3
y[ 2] = 2.2

x[ 3] = 4
y[ 3] = 2.5

Iterpolalando ertek (xh): 2.5

Lagrange-interpolacio
Interpolacio helye(xh): 2.500000
Intepolalt ertek (yh) : 2.031250
```

12.4 Lineáris egyenletrendszer megoldása

Számos probléma megoldása visszavezethető lineáris egyenletrendszer megoldására. A gyakorlatban direkt és iteratív módszereket egyaránt alkalmazunk. A következőkben a lineáris egyenletrendszer megoldásának fontosabb módszereit ismertetjük.

A lineáris egyenletrendszer

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,n}x_n &= y_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \ldots + a_{2,n}x_n &= y_2 \\ &\ldots \\ &\ldots \\ &\ldots \\ a_{n,1}x_1 + a_{n,2}x_2 + \ldots + a_{n,n}x_n &= y_n \end{aligned}$$

mátrix írásmódban

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \ldots & a_{1,n} \\ a_{2,1} & a_{2,2} & \ldots & a_{2,n} \\ \ldots & & & \\ a_{n,1} & a_{n,2} & \ldots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \ldots \\ y_n \end{bmatrix}$$

$$\mathbf{A}\,\mathbf{x} = \mathbf{y}$$

A feladat az **x** vektor meghatározása. Feltételezzük, hogy minden ismeretlennek legalább egy együtthatója nem nulla, és hogy az egyenlet együtthatóinak determinánsa nem nulla. Ilyenkor az egyenletrendszernek létezik megoldása:

$$x_1 \quad x_2 \quad \ldots \quad x_n .$$

Amennyiben az egyenletrendszer determinánsa nulla, akkor az egyenletrendszernek általában nincs megoldása, de előfordulhat, hogy végtelen sok megoldása létezik. Eb-

ben az esetben egy vagy több ismeretlen értéke szabadon megválasztható, a többi ismeretlen pedig ezek lineáris függvénye.

A lineáris egyenletrendszerek megoldása különböző módszerekkel történhet
- kiküszöbölési eljárással (direkt módszer),
- fokozatos közelítés módszerével (iteratív módszer).

Az **y** jobb oldali vektort helyezzük el az **A** mátrix n+1-edik oszlopába. Ha az **A** mátrixot egy oszloppal kiterjesztjük, könnyebb az algoritmus kidolgozása

$$a_{i,n+1} = y_i$$

Az egyenlet ekkor az alábbi alakot veszi fel:

$$\sum_{j=1}^{n} a_{i,j} \cdot x_j = a_{i,n+1} \qquad (i = 1,2,\ldots n)$$

12.4.1 Lineáris egyenletrendszer megoldása LU-dekompozícióval

Gauss-eliminációval végzett lineáris egyenletrendszer megoldás során a jobb oldal is transzformálódik:

$$\mathbf{A'\,x = B'}$$

Ha az egyenletrendszert több jobb oldallal kell megoldani úgy, hogy ezek kezdetben még nem mind állnak rendelkezésre, akkor alkalmazzuk az LU dekompozíciót, mert ennél a módszernél a nagy műveletigényű dekompozíciót egyszer végezzük el, így akárhány jobb oldallal is megoldhatjuk az egyenletrendszert.

Az **A** mátrixot felbontjuk két háromszögmátrixra **L** (*Lower* = alsó) és egy **U** (*Upper* = felső) mátrixra:

$$\mathbf{A\,x = B}$$

Behelyettesítve az **A** mátrix helyére az **L U** mátrixszorzatot:

$$\mathbf{L\,U\,x = B}$$

Ahol az **L** mátrix (elemeit r-rel jelöljük), illetve az **U** mátrix (elemeit u-val jelöljük)

$$\begin{bmatrix} 1 & 0 & . & 0 & 0 \\ r_{2,1} & 1 & . & 0 & 0 \\ . & . & 1 & 0 & 0 \\ . & . & . & 1 & 0 \\ r_{n,1} & r_{n,2} & . & r_{n,n-1} & 1 \end{bmatrix} \qquad \begin{bmatrix} u_{1,1} & u_{1,2} & . & . & u_{1,n} \\ 0 & u_{2,2} & . & . & u_{2,n} \\ . & . & . & . & . \\ 0 & 0 & 0 & u_{n-1,n-1} & u_{n-1,n} \\ 0 & 0 & 0 & 0 & u_{n,n} \end{bmatrix}$$

Az *n*-ed rendű LU mátrix *n-1* iteráció eredményeként jön létre. Az iteráció folyamán először a főátlóbeli elem alatti elemeket osztjuk a főátlóbeli elemmel, majd a mátrix további elemeit módosítjuk a módosított elemekkel az alábbi módon:

$$a_{i,k}^{(k)} = \frac{a_{i,k}^{(k-1)}}{a_{k,k}^{(k-1)}} \qquad k = 1,2, \ldots n-1$$

$$a_{i,j}^{(k)} = a_{i,j}^{(k-1)} - a_{i,k}^{(k)} \cdot a_{k,j}^{(k-1)} \qquad \begin{matrix} i = k+1, \ldots n \\ j = k+1, \ldots n \end{matrix}$$

Az iteráció eredményeként a két mátrix (**L** és **U**) együttesen tárolásra kerül, a főátlóban az *u* mátrix elemei szerepelnek.

$$\begin{bmatrix} u_{1,1} & u_{1,2} & . & . & u_{1,n} \\ r_{2,1} & u_{2,2} & . & . & u_{2,n} \\ . & . & . & . & . \\ r_{n-1,1} & . & . & u_{n-1,n-1} & u_{n,n-1} \\ r_{n,1} & r_{n,2} & . & r_{n,n-1} & u_{n,n} \end{bmatrix}$$

Az LU-dekompozíció bemutatása általánosan

Az **A** 4x4-es mátrix LU-dekompozícióját 3 (*n-1*) iterációs lépés után kapjuk meg. Általánosan a feladat megoldása a következő:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix}$$

Az **A** mátrix elemeinek módosulása az első iterációs lépés után (*k = 1*):

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ \frac{a_{2,1}}{a_{1,1}} & a_{2,2} - \frac{a_{2,1}}{a_{1,1}} \cdot a_{1,2} = a'_{2,2} & a_{2,3} - \frac{a_{2,1}}{a_{1,1}} \cdot a_{1,3} = a'_{2,3} & a_{2,4} - \frac{a_{2,1}}{a_{1,1}} \cdot a_{1,4} = a'_{2,4} \\ \frac{a_{3,1}}{a_{1,1}} & a_{3,2} - \frac{a_{3,1}}{a_{1,1}} \cdot a_{1,2} = a'_{3,2} & a_{3,3} - \frac{a_{3,1}}{a_{1,1}} \cdot a_{1,3} = a'_{3,3} & a_{3,4} - \frac{a_{3,1}}{a_{1,1}} \cdot a_{1,4} = a'_{3,4} \\ \frac{a_{4,1}}{a_{1,1}} & a_{4,2} - \frac{a_{4,1}}{a_{1,1}} \cdot a_{1,2} = a'_{4,2} & a_{4,3} - \frac{a_{4,1}}{a_{1,1}} \cdot a_{1,3} = a'_{4,3} & a_{4,4} - \frac{a_{4,1}}{a_{1,1}} \cdot a_{1,4} = a'_{4,4} \end{bmatrix}.$$

Az **A** mátrix elemeinek módosulása a második iterációs lépés után ($k = 2$):

$$\begin{bmatrix} a_{1,1} & a_{2,1} & a_{1,3} & a_{1,4} \\ \frac{a_{2,1}}{a_{1,1}} & a'_{2,2} & a'_{2,3} & a'_{2,4} \\ \frac{a_{3,1}}{a_{1,1}} & \frac{a'_{3,2}}{a'_{2,2}} = a''_{3,2} & a'_{3,3} - a''_{3,2} \cdot a'_{2,3} = a''_{3,3} & a'_{3,4} - a''_{3,2} \cdot a'_{2,4} = a''_{3,4} \\ \frac{a_{4,1}}{a_{1,1}} & \frac{a'_{4,2}}{a'_{2,2}} = a''_{4,2} & a'_{4,3} - a''_{4,2} \cdot a'_{2,3} = a''_{4,3} & a'_{4,4} - a''_{4,2} \cdot a'_{2,4} = a''_{4,4} \end{bmatrix}$$

Az **A** mátrix elemeinek módosulása a harmadik iterációs lépés után ($k = 3$):

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ \frac{a_{2,1}}{a_{1,1}} & a'_{2,2} & a'_{2,3} & a'_{2,4} \\ \frac{a_{3,1}}{a_{1,1}} & \frac{a'_{3,2}}{a'_{2,2}} & a''_{3,3} & a''_{3,4} \\ \frac{a_{4,1}}{a_{1,1}} & \frac{a'_{4,2}}{a'_{2,2}} & \frac{a''_{4,3}}{a''_{3,3}} = a'''_{4,3} & a''_{4,4} - a'''_{4,3} \cdot a''_{3,4} = a'''_{4,4} \end{bmatrix}$$

$$\mathbf{A\,x = L\,U\,x = L\,\underbrace{(U\,x)}_{\mathbf{y}} = L\,y = b}$$

Először oldjuk meg az

$$\mathbf{L\,y = b}$$

lineáris egyenletrendszert!

$$\begin{bmatrix} 1 & 0 & 0 & . & 0 \\ r_{2,1} & 1 & . & . & 0 \\ . & . & 1 & 0 & 0 \\ . & . & . & 1 & 0 \\ r_{n,1} & r_{n,2} & . & r_{n,n-1} & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ . \\ . \\ b_n \end{bmatrix}$$

Az egyenletből határozzuk meg az **y** vektort. Az első egyenletből kiindulva

$$y_1 = b_1$$

a további ismeretlenek meghatározása:

$$y_k = b_k - \sum_{j=1}^{k-1} r_{k,j} \cdot y_j, \qquad k = 2, 3, \ldots n$$

Majd oldjuk meg az

$$\mathbf{U}\,\mathbf{x} = \mathbf{y}$$

lineáris egyenletrendszert! Az **y** vektor ismeretében az **x** vektor meghatározható.

$$\begin{bmatrix} u_{1,1} & u_{1,2} & . & . & u_{1,n} \\ 0 & u_{2,2} & . & . & u_{2,n} \\ 0 & 0 & . & . & . \\ 0 & . & . & u_{n-1,n-1} & u_{n-1,n} \\ 0 & . & . & 0 & u_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ y_n \end{bmatrix}$$

Az utolsó egyenletből kiindulva:

$$x_n = \frac{y_n}{u_{n,n}}$$

a további ismeretlenek meghatározása:

$$x_k = \frac{y_k - \sum_{j=k+1}^{n} u_{k,j} \cdot x_j}{u_{k,k}}, \qquad k = n-1, n-2, \ldots 1$$

Az LU-dekompozíció numerikus stabilitásához szükséges legalább a részleges főelem-kiválasztás (*dekompf* függvény).

Oldjuk meg az alábbi egyenletrendszert egyszerű és a részleges főelem-kiválasztással működő LU-dekompozícióval! (*LU*)

$$10x_1 + 5x_2 + 2x_3 = 9$$
$$20x_1 + 12x_2 + 7x_3 = 22$$
$$30x_1 + 19x_2 + 17x_3 = 45$$

```
/*MAIN.C*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>

#define MAXELEM 20
typedef double tomb[MAXELEM+1][MAXELEM+2];
typedef double vekt[MAXELEM+1];

void olvas(tomb a, tomb e, int *n);
```

```
void eredmeny(tomb a, tomb e, vekt x, int d, int n);
void matrixkiir(tomb a, int n);
void sorcsere(tomb a, tomb e, int xm, int k, int n);
void foelem(tomb a, tomb e, int n, int k, int *d);
void dekomp(tomb a, int n,int *d);
void dekompf(tomb a, tomb e, int n,int *d);
void yvektor(vekt y,tomb a, int n);
void xvektor(vekt x, vekt y, tomb a, int n);
void inverz(tomb a, tomb e, int n);
double det(tomb a, int n, int d);

int main()
{  int  n=0, d=1, v=0;
   tomb a, e;
   vekt x, y;
   printf("\nA x = b linearis egyenletrendszer megoldasa\n");
   printf("          LU-dekompozicioval\n\n");
   printf("1.  Foelem-kivalasztas nelkul\n");
   printf("2.  Reszleges-foelemkivalasztassal\n");
   printf("3.  Kilepes");
   do{
      printf("\n\nMenu kivalasztasa: ");
      scanf("%d",&v);
   } while ( v > 3 || v < 1);
   switch(v)
   {
    case 1: olvas(a,e,&n);
            printf("\nA matrix \n\n");
            matrixkiir(a,n);
            dekomp(a,n,&d);
            break;
    case 2:
            olvas(a,e,&n);
            printf("\nA matrix \n\n");
            matrixkiir(a,n);
            dekompf(a,e,n,&d);
            break;
    case 3: return 0;
   }
   yvektor(y,a,n);
   xvektor(x,y,a,n);
   eredmeny(a,e,x,d,n);
   fflush(stdin); getchar();getchar();
   return 0;
}

void olvas(tomb a,tomb e, int *n)
{ int i, j;
  do {
       printf("\nElemek szama [max. 20]: ");
       scanf("%d",n);
  }while ( *n > MAXELEM);
  printf("\n");
```

```
  for ( i = 1; i <= *n; i++ )
  {
    for ( j = 1; j <= *n; j++ )
    {
      printf("a[%d,%d]= ",i,j);
      scanf("%lf",(a[i]+j));
      if ( i == j )
      {
       e[i][j] = 1.0;
      }
      else
      {
        e[i][j] = 0.0;
      }
    }
    printf("\n");
  }
  for ( i = 1; i <= *n; i++ )
  {
    printf("b[%d]= ",i);
    scanf ("%lf",&a[i][*n+1]);
  }
}

/* Az eredmények kiíratása */
void eredmeny(tomb a, tomb e, vekt x, int d, int n)
{ int i;
   printf("\nAz egyenlet gyokei\n\n");
   for ( i=1; i<=n; i++ )
   {
     printf("x[%d]=  %.4lf\n",i,x[i]);
   }

   printf("\nAz inverz matrix:\n\n");
   inverz(a,e,n);

   printf("\nA determinans: %lf\n",det(a,n,d));
}

void matrixkiir(tomb a, int n)
{ int i,j;
  for ( i=1; i<=n; i++ )
  {
    for ( j=1; j<=n; j++)
    {
      printf("%10.3lf",a[i][j]);
    }
    printf("\n");
  } printf("Barmely karakter leutese a folytatashoz...\n"); getch();
}

/* A mátrix sorainak felcserélése  */
void sorcsere(tomb a,tomb e, int xm, int k, int n)
{ int i;
  double c;
```

```
  for ( i = 1; i <= n+1 ; i++ )
  {
    c=a[xm][i]; a[xm][i]=a[k][i]; a[k][i]=c;
    c=e[xm][i]; e[xm][i]=e[k][i]; e[k][i]=c;
  }
}
/* A főelem-kiválasztása */
void foelem(tomb a,tomb e,int n,int k, int *d)
{ int i,xm;
  double am;
  am=a[k][k]; xm=k;
  for ( i = k; i <= n; i++ )
  {
    if (fabs(am) < fabs(a[i][k]))
    {
      am=a[i][k]; xm=i;
      sorcsere(a,e,xm,k,n);
      *d=*d *(-1);
    }
  }
}
/* dekompozició végrehajtása */
void dekomp(tomb a, int n, int *d)
{  int i,j,k;
  *d = 1;
   for ( k = 1; k < n; k++ )
   {  printf("\n%d. iteracio Oszt: %10.3lf\n\n",k,a[k][k]);
      for ( i = k+1; i <= n; i++ )
      {
        a[i][k]=a[i][k]/a[k][k];
        for ( j = k+1; j <= n; j++ )
        {
          a[i][j]=a[i][j]-a[i][k]*a[k][j];
        }
      }
      matrixkiir(a,n);
   }
}
/* dekompozició végrehajtása főelem-kiválasztásával */
void dekompf(tomb a,tomb e, int n, int *d)
{
   int i,j,k;
   *d = 1;
   for ( k = 1; k < n; k++ )
   {  foelem(a,e,n,k,d);
      printf("\n%d. iteracio Oszt: %10.3lf\n\n",k,a[k][k]);
      for ( i = k+1; i <= n; i++ )
      {
        a[i][k]=a[i][k]/a[k][k];
        for ( j = k+1; j <= n; j++ )
        {
          a[i][j]=a[i][j]-a[i][k]*a[k][j];
        }
      }
```

```
      matrixkiir(a,n);
    }
}
/*  Ly = b egyenlet megoldása */
void yvektor(vekt y,tomb a,int n)
{ int j,k;
  y[1]=a[1][n+1];
  for ( k = 2; k <= n; k++)
  {
    y[k]=0;
    for ( j = 1; j <= k-1; j++)
    {
      y[k]=y[k]-a[k][j]*y[j];
    }
    y[k]=y[k]+a[k][n+1];
  }
}
/*  U x = y egyenlet megoldása */
void xvektor(vekt x,vekt y,tomb a,int n)
{   int j,k;
    x[n]=y[n]/a[n][n];
    for ( k = n-1; k >= 1; k--)
    {
      x[k]=0;
      for ( j = k+1; j <= n; j++ )
      {
       x[k]=x[k]-a[k][j]*x[j];
      }
      x[k]=(y[k]+x[k])/a[k][k];
    }
}
/* A mátrix inverzének kiszámítása */
void inverz(tomb a, tomb e, int n)
{ int i,j;
  vekt x, y;
  for ( j = 1; j <= n; j++ )
  {
    for ( i = 1; i <= n; i++ )
    {
      a[i][n+1]=e[i][j];
    }
    yvektor(y,a,n);
    xvektor(x,y,a,n);
    for ( i = 1; i <= n; i++ )
    {
      e[i][j]=x[i];
    }
  }
  matrixkiir(e,n);
}
```

```
/* A determináns kiszámítása */
double det(tomb a, int n, int d)
{ int k;
  double c;
  c=a[1][1];
  for ( k = 2; k <= n; k++ )
  {
    c=c*a[k][k];
  }
  return (d * c);
}
```

A program futásának eredménye:

```
A x = b linearis egyenletrendszer megoldasa
          LU-dekompozicioval

1.  Foelem-kivalasztas nelkul
2.  Reszleges foelem-kivalasztassal
3.  Kilepes

Menu kivalasztasa: 1

Elemek szama [max. 20]: 3

a[1,1]= 10
a[1,2]= 5
a[1,3]= 2

a[2,1]= 20
a[2,2]= 12
a[2,3]= 7

a[3,1]= 30
a[3,2]= 19
a[3,3]= 17

b[1]= 9
b[2]= 22
b[3]= 45

A matrix

    10.000     5.000     2.000
    20.000    12.000     7.000
    30.000    19.000    17.000
Barmely karakter leutese a folytatashoz...

1. iteracio Oszt:     10.000

    10.000     5.000     2.000
     2.000     2.000     3.000
     3.000     4.000    11.000
Barmely karakter leutese a folytatashoz...

2. iteracio Oszt:      2.000

    10.000     5.000     2.000
     2.000     2.000     3.000
     3.000     2.000     5.000
Barmely karakter leutese a folytatashoz...
```

```
Az egyenlet gyokei

x[1]=  1.0000
x[2]=  -1.0000
x[3]=  2.0000

Az inverz matrix:

     0.710     -0.470      0.110
    -1.300      1.100     -0.300
     0.200     -0.400      0.200
Barmely karakter leutese a folytatashoz...

A determinans: 100.000000
```

12.4.2 Mátrixinvertálás

Ha az **A** mátrixnak **B** mátrix az inverze, akkor a

$$\mathbf{B}\,\mathbf{A} = \mathbf{E} \quad \text{és} \quad \mathbf{A}\,\mathbf{B} = \mathbf{E}$$

szorzatok egységmátrixot kell, hogy adjanak.
Ahol az **E** egységmátrix

$$E = \begin{bmatrix} 1 & 0 & . & . & 0 \\ 0 & 1 & . & . & 0 \\ . & . & 1 & . & 0 \\ 0 & 0 & . & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Mátrixinvertálás LU dekompozícióval

A mátrix inverzének meghatározásához az egyszer dekomponált LU mátrixot n-szer megoldjuk az **E** egységmátrix oszlopvektoraival.

$$\mathbf{L}\,\mathbf{U}\,\mathbf{x} = e_i \quad \text{ahol} \quad i = 1,2, \ldots n$$

$$e_1 = \begin{bmatrix} 1 \\ 0 \\ . \\ . \\ 0 \end{bmatrix} \qquad e_2 = \begin{bmatrix} 0 \\ 1 \\ . \\ . \\ 0 \end{bmatrix} \quad \cdots \quad e_n = \begin{bmatrix} 0 \\ 0 \\ . \\ . \\ 1 \end{bmatrix}$$

A kapott gyökök vektorai szolgáltatják az invertált mátrix oszlopvektorait. A sorozatosan kapott eredményt betöltjük a **B** mátrix oszlopaiba, és az n-edik megoldás után előáll a **B** mátrixban az **A** mátrix inverze.

```
. . .
int main()
{
   int  n, d=1;
   tomb a, e;
   printf("\nMatrix invertalasa LU-dekompozicioval\n");
   printf("   reszleges foelem-kivalasztassal \n\n");
   do {
       printf("\nElemek szama [max. 20]: ");
       scanf("%d",&n);
   }while ( n > MAXELEM);

   olvas(a,e,n);
   printf("\nA matrix \n\n");
   matrixkiir(a,n);
   dekompf(a,e,n,&d);
   eredmeny(a,e,d,n);
   fflush(stdin); getchar();getchar();
   return 0;
}

. . .
/* A mátrix inverzének kiszámítása */
void inverz(tomb a,tomb e, int n)
{
  int i,j;
  vekt x, y;
  for ( j = 1; j <= n; j++ )
  {
    for ( i = 1; i <= n; i++ )
    {
      a[i][n+1]=e[i][j];
    }
    yvektor(y,a,n);
    xvektor(x,y,a,n);
    for ( i = 1; i <= n; i++ )
    {
      e[i][j]=x[i];
    }
  }
  matrixkiir(e,n);
}
/* A determináns kiszámítása */
double det(tomb a, int n, int d)
{ int k;
  double c;
  c=a[1][1];
  for ( k = 2; k <= n; k++ )
  {
    c=c*a[k][k];
  }
  return (d * c);
}
```

A további függvények az LU programból más ismertek.

Invertáljuk LU-dekompozícióval az alábbi mátrixot! (*INVERZ*)

$$\begin{bmatrix} 0{,}0001 & 1 & 1 \\ 1 & 0{,}00001 & 1 \\ 1 & 4 & 0{,}00001 \end{bmatrix}$$

A program futásának eredménye:

```
Matrix invertalasa LU-dekompozicioval
   reszleges foelem-kivalasztassal

Elemek szama [max. 20]: 3

a[1,1]= 0.0001
a[1,2]= 1
a[1,3]= 1

a[2,1]= 1
a[2,2]= 0.00001
a[2,3]= 1

a[3,1]= 1
a[3,2]= 4
a[3,3]= 0.00001

. . .

Az inverz matrix:

 -8.0007e-001  8.0007e-001  2.0001e-001
  2.0001e-001 -2.0002e-001  2.0000e-001
  8.0007e-001  1.9994e-001 -2.0002e-001
Barmely karakter leutese a folyatashoz...

A determinans: 4.999580e+000
```

F1. A Dev-C++ fejlesztői környezet használata

F1.1 A C program állományai

A C programot egyetlen állomány (.C) is tartalmazhatja, azonban ha több modulból áll, a fájlokat a projekt fogja össze.

A feladatok megoldása során általában az alábbiakat hoztuk létre:

- Egy állományból álló C program:
  - A ***main()*** függvényt és egyéb függvényeket tartalmazó *MAIN.C* állomány.
  - A futtatható program neve azonos az állomány nevével, *MAIN.EXE.*

- Több fájlból felépülő C program:
  - A ***main()*** függvényt tartalmazó *MAIN.C* állomány.
  - Egyéb függvényeket tároló .C állományok.
  - Függvények prototípusát és deklarációkat tartalmazó .H állományok.
  - A program állományait a *PROJECT1.DEV* fogja össze.
  - A futtatható program neve azonos a projekt nevével (például *PROJECT1.EXE).*

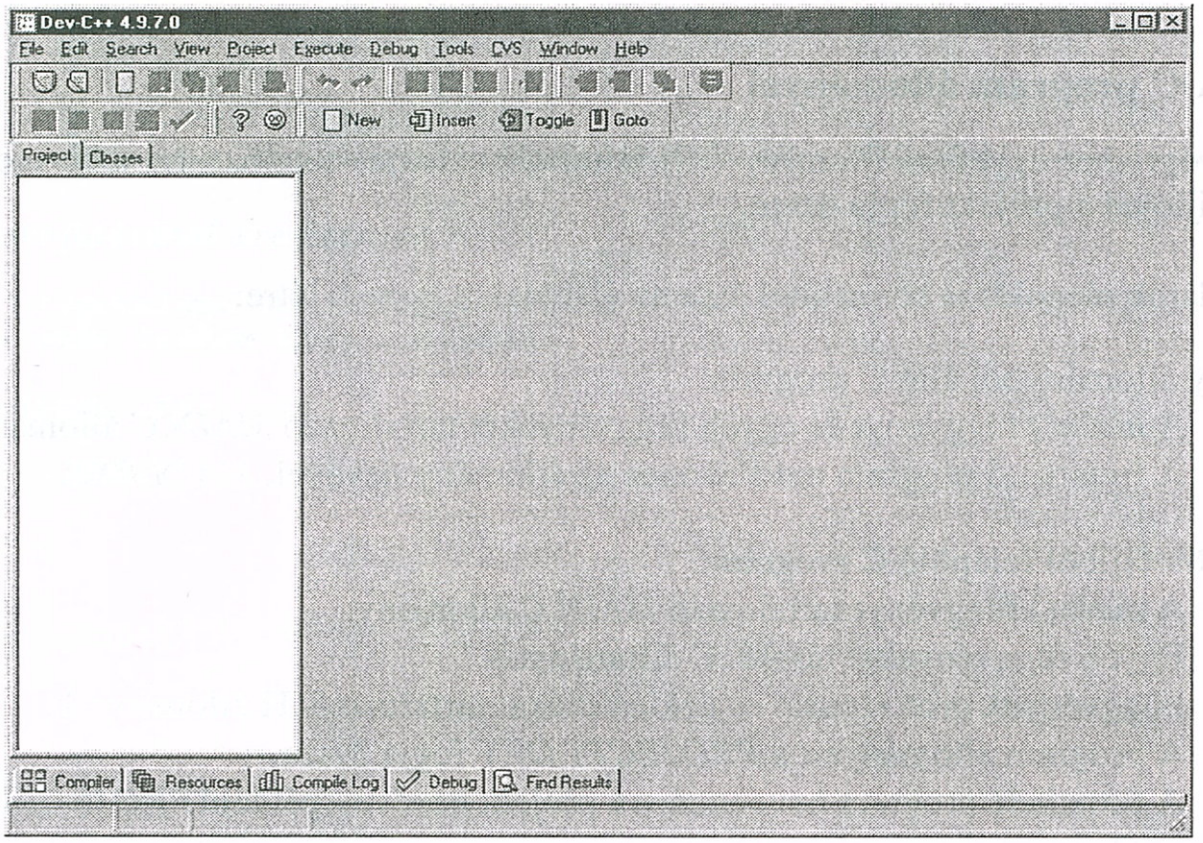
F1.2 Új forrásállomány és project létrehozása

Először telepítsük a CD-mellékleten található 4.9.7.0 verziójú, szabad felhasználású Dev-C++ fejlesztői környezetet, majd tegyük ki az Asztalra a parancsikonját! Ehhez keressük meg az *Intézővel* a *Dev-Cpp* könyvtárban a *devcpp* állományt!

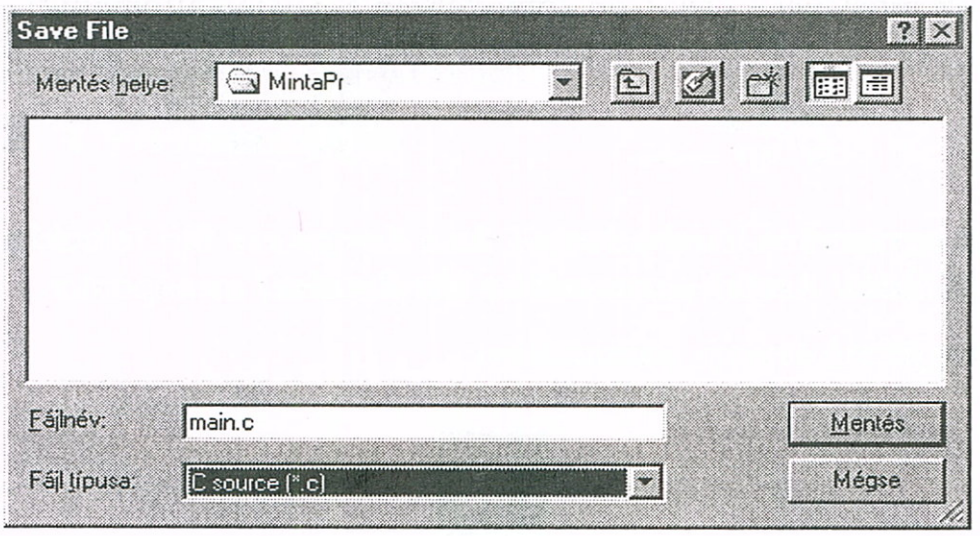
A parancsikont egyszerűen létrehozhatjuk, ha a *devcpp.exe* fájl felbukkanó menüjéből kiválasztjuk a **Küldés/ Asztal(parancsikon létrehozása)** menüpontot.

Az *Asztalon* megjelenő *devcpp* parancsikon:

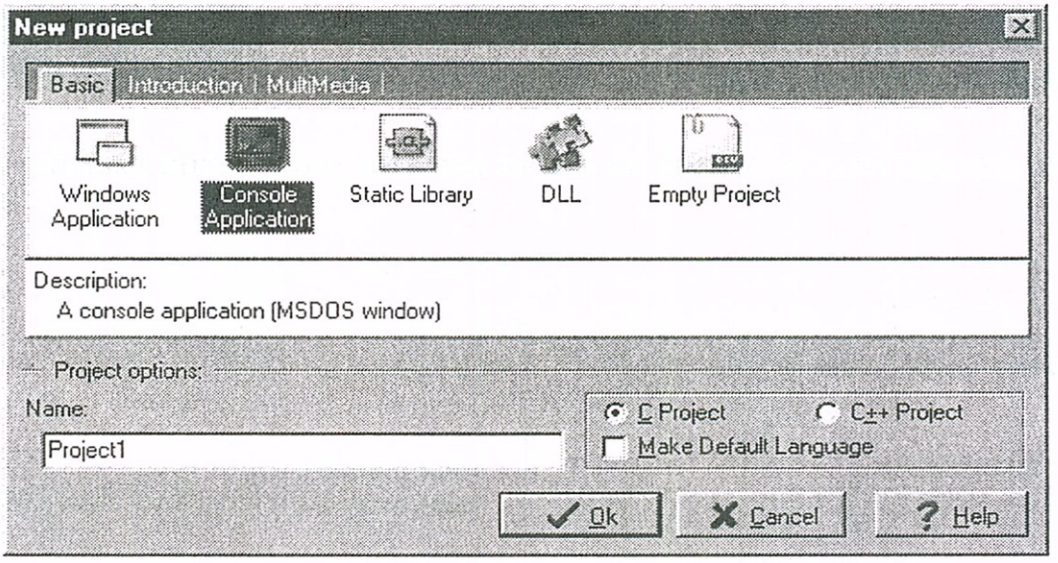
A *devcpp* ikonon kattintva a fejlesztői környezet ablaka válik láthatóvá.



Egyetlen állományból álló C programot a **File/New/Source File** menüpont kiválasztásával vagy a *<Ctrl+N>* gyorsítóbillentyű megnyomásával hozhatunk létre például a *MINTAPR* könyvtárban *MAIN.C* néven.



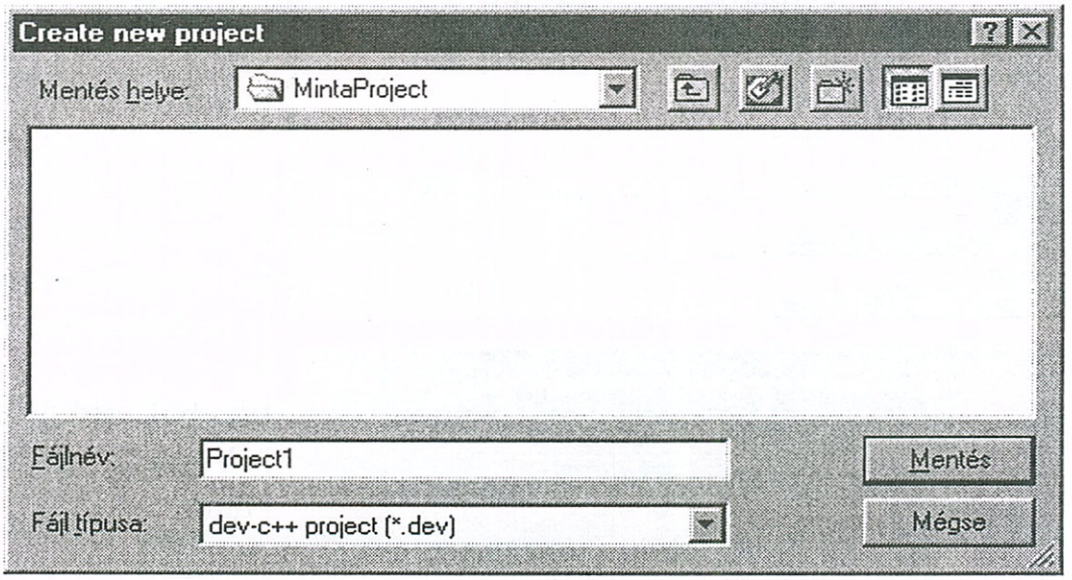
A több állományból álló C programhoz a **File/New/Project...** menüpont kiválasztásával új projektet készíthetünk, ha a megjelenő párbeszédablakban kijelöljük a *Console Application* ikont, majd a *C Project* választógombot bejelölve kattintunk az *OK* nyomógombra.



Példaként bemutatjuk a *MintaProject* feladat projektjének felépítését. A *Project1* állomány számára *MintaProject* néven hozzunk létre egy könyvtárat az *Új mappa létrehozása* ikonon kattintva!

Új mappa létrehozás

A létrehozott könyvtár nevén kattintva belépünk a könyvtárba, és a felajánlott *Project1* nevet elfogadva, megnyomjuk a *Mentés* nyomógombot.



A **File/New/Source File** menüpont kiválasztásával vagy *<Ctrl+N>* gyorsítóbillentyű megnyomásával új forrásállományt adhatunk az aktuális *projekt-hez.*

A forrásállomány *Untitled1* néven jön létre, melyet a **File/Save As...** menüpont kiválasztásával *FUGGVENY.C* fájlként mentünk el. (Szükség esetén létrehozhatunk még további állományokat, példaként egy FUGGVENY.H állományt is.)

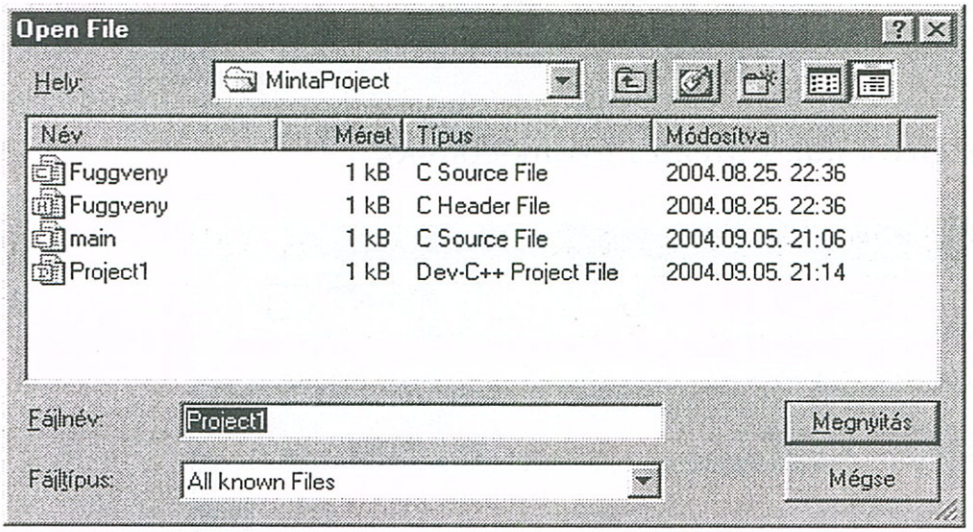
A ***main()*** függvény, illetve a függvényt tároló .C fájl automatikusan létrejön, melyet a **File/Save As...** menüpont kiválasztásával *MAIN.C* állományként lemezre mentünk.

A felajánlott programot módosítjuk az alábbi táblázat jobb oldala szerint:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{

  system("PAUSE");
  return 0;
}
```

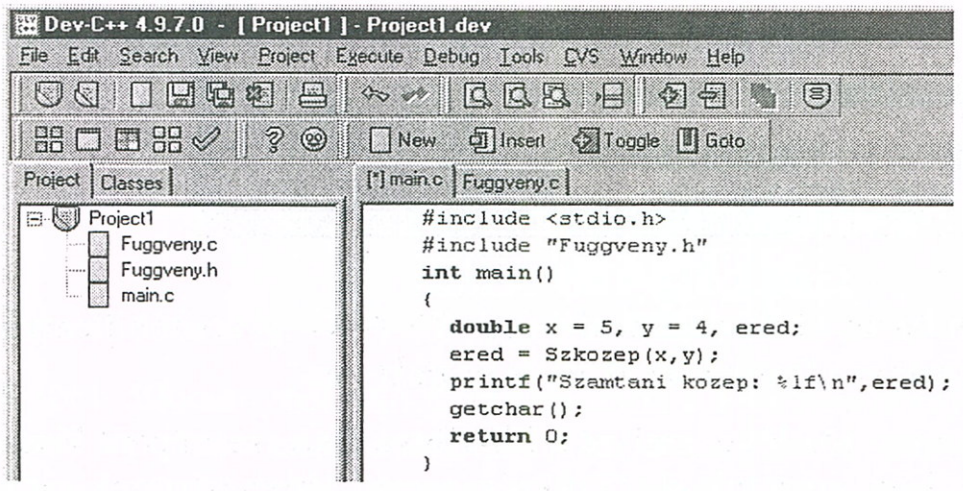
```
#include <stdio.h>


int main()
{

  getchar();
  return 0;
}
```

A *MintaProject* könyvtár szerkezete és tartalma:



A *MintaProject* könyvtár állományai:

FUGGVENY.C
FUGGVENY.H
MAIN.C
PROJECT1.DEV



A bal oldali ablakban a *Project1* szerkezete látható.

A jobb oldali ablakban pedig a *MAIN.C* állományban lévő ***main()*** függvény definíciója jelenik meg.

Név	Méret
..	<UP-DIR>
Fuggveny.c	61
Fuggveny.h	53
Fuggveny.o	451
main.c	186
main.o	643
Makefile.win	623
Project1.dev	951
Project1.exe	10 037
Project1.layout	219

A lefordítás után a program könyvtárában a *PROJECT1.EXE* a futtatható állomány.

F1.3 A program futtatása

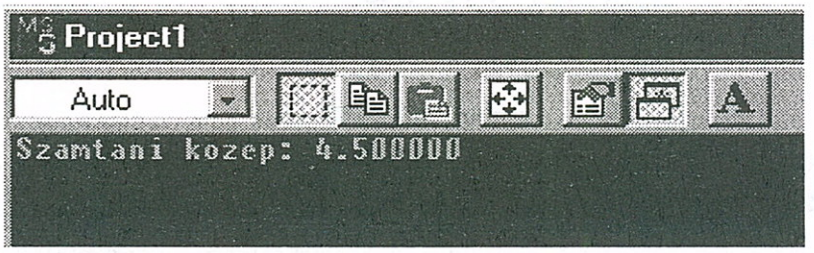
A programot az **Execute/Compile & Run** menüpont kiválasztásával vagy az *<F9>* gyorsítóbillentyű segítségével futtathatjuk.

Az eszköztár gombjaival a következő műveleteket végezhetjük el:

	Compile <Ctrl+F9>	Fordítás.
	Run <Ctrl+F10>	Futtatás.
	Compile & Run <F9>	Fordítás és futtatás.
	Rebuild All <Ctrl+F11>	A program újrafordítása.

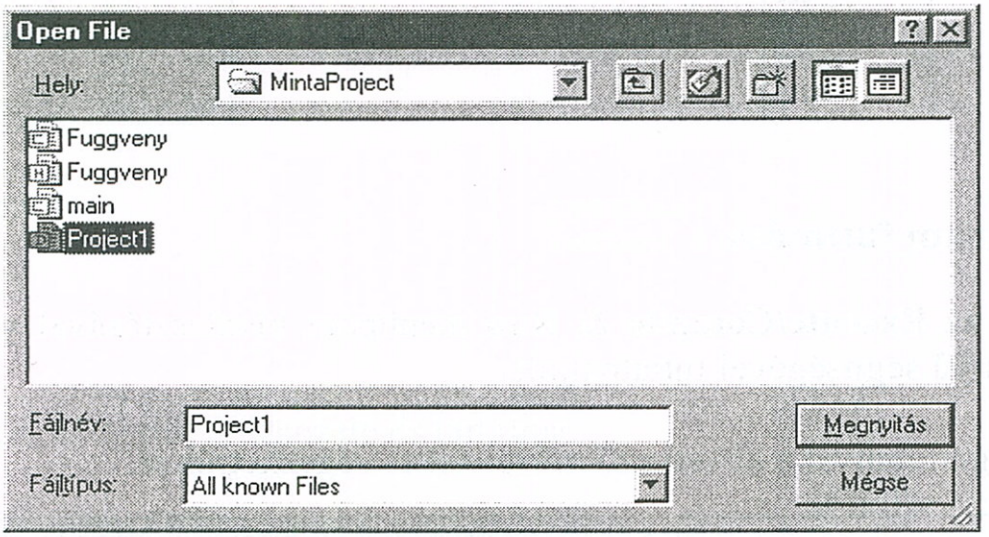
Ha biztosak akarunk lenni abban, hogy a program minden állománya újra lefordul, akkor használjuk az **Execute/Rebuild All** menüpontot, vagy *<Ctrl+F11>* gyorsítóbillentyűt! A program az újrafordítás után nem indul el automatikusan, az indítás az **Execute/Run** menüpont kiválasztásával, vagy *<Ctrl+F10>* gyorsító billentyűk megnyomásával végezhető el.

A program futásának eredménye a projekt nevének megfelelő fejlécű ablakban jelenik meg:



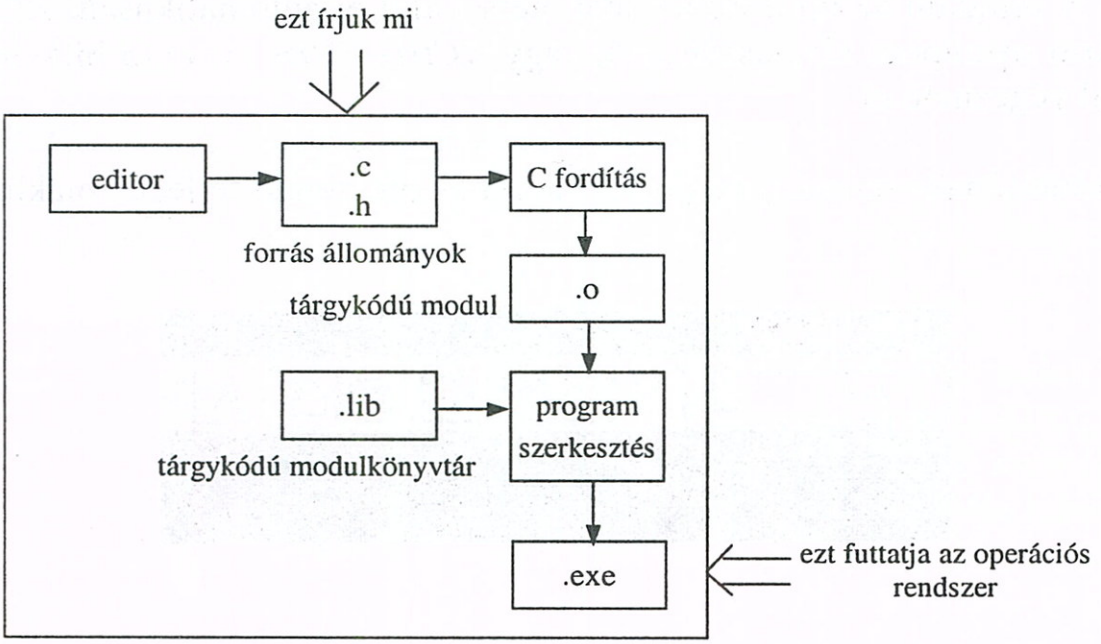
F1.4 Project betöltése

A **File/Open Project or File...** menüpont kiválasztásával, vagy *<Ctrl+O>* gyorsítóbillentyű megnyomásával megkeressük a könyvtárat, kijelöljük a *Project1* fájlnevet és a *Megnyitás* nyomógombon kattintva a program állományai betöltődnek.



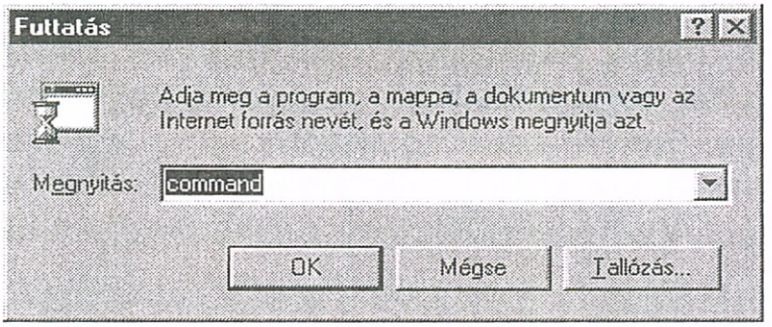
F1.5 Hogyan működik a fejlesztői környezet

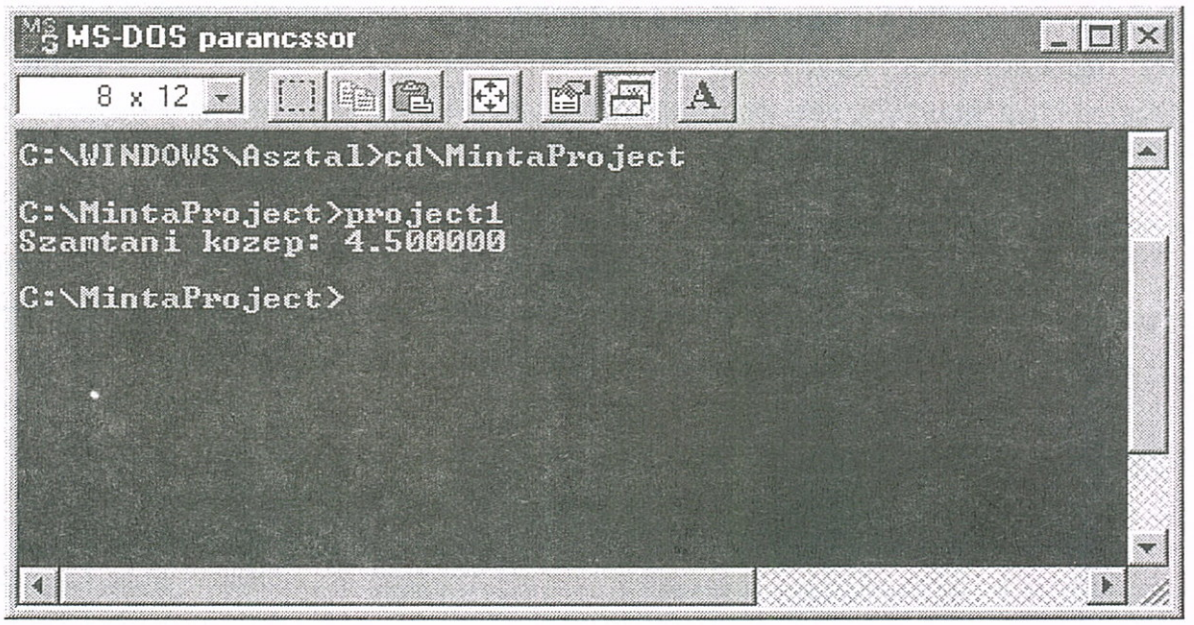
A fejlesztői környezet azt a kényelmi szolgáltatást nyújtja a számunkra, hogy a szövegszerkesztőjében megírt C forrásprogramból automatikusan előállítja az operációs rendszer által futtatható, kész programot és le is futtatja azt. A kész programjainkat egy ún. konzol ablakban futtatja, ami csak addig van nyitva, amíg az .EXE programunk fut, így a program szöveges kimenete is csak addig létezik.



E probléma megoldására két lehetőség van:

- Az egyik (és ez az általában használandó módszer) az, hogy mi magunk nyitunk az operációs rendszerben egy ún. parancsértelmező konzol ablakot (**Start/Futtatás**: **command**) és ebből az ablakból kézzel elindítjuk a kész .EXE programunkat. Ennek szöveges kimenete a parancsértelmező konzol ablakban látható lesz.





- A másik módszer az STDIO.*H* fejállományban deklarált ***getchar()*** függvény használata. E könyvben mi ezt használtuk, hogy a kész programok futtatásához ne kelljen a fejlesztői környezeten kívül mást is használni. Ha viszont a program adatot olvas be, akkor az inputpufferben lévő *<Enter>*-t először az ***fflush***(*stdin*) hívásával töröljük és a ***getchar()*** pedig felfüggeszti a programot az újabb *<Enter>* billentyű leütéséig.

  Megjegyezzük azonban, ha a feladat az, hogy a programunk egy adatfolyamot szűrőszerűen dolgozzon fel, akkor ezt a feldolgozási műveletet a program végén lévő ***getchar()*** kiírás megzavarhatja.

F2. C óravázlatok

Az F2. függelékben olyan oktatási anyag található, amely 11 gyakorlati órára osztja az alap C nyelv témaköreit. Az oktatási anyag bővebb, magyarázatokkal ellátott és alkalmazási példákkal kiegészített feladatlapjai PDF-formátumban megtalálhatók a CD-melléklet *Tanari kezikonyv* nevű könyvtárában. A feladatsorok felölelik a könyv teljes tartalmát, azonban az alapkönyv feladataitól eltérő feladatokon keresztül mutatják be a C nyelv lehetőségeit, és lépésről-lépésre tárják az Olvasó elé az ismereteket. A gyakorlatok elején néhány gondolat bevezeti az alfejezet témakörét, illetve a hozzá tartozó mintapéldákat. A mintapéldák kidolgozott megoldásának célja, hogy segítse a további feladatok önálló, gördülékenyebb megoldását.

Az óravázlatokhoz tartozó forráskódok a gyakorlati foglalkozás sorszámának megfelelően alkönyvtárakba kerültek. Például, az első gyakorlat programjainak forráskódját a *GYAK01_C* alkönyvtár tartalmazza. Minden C alkalmazást külön könyvtárban tárolunk, a gyakorlati foglalkozás és a feladat sorszámának megfelelően. Például, a *GYAK1_1* alkönyvtár az első gyakorlat 1. feladatának megoldását tartalmazza, míg a *MINTA1_01* mappában az 1. gyakorlat 1. mintaprogramjának forráskódja található.

Gyakorlat	A gyakorlat tematikája
1.	C nyelv alapelemei, jelkészlete. Konstansok, megjegyzések. Egyszerű változók definiálása. Operátorok és kifejezések. A C program szerkezete. Szabványos input és output adatfolyamok elérése a ***scanf()***, illetve a ***printf()*** könyvtári függvényekkel, melyek prototípusát az *STDIO.H* fejállomány tartalmazza.
2.	Összehasonlító, logikai és léptető operátorok. Bitléptető operátorok, biteltoló műveletek, és a **sizeof** operátor. A C++ nyelv utasításai. Az **if** feltételes utasítás. A **switch** többirányú elágaztatás.
3.	Ciklusutasítások: **for**, **while**, **do-while**. A **continue** és a **break** utasítások.
4.	Mutatók és többszörös indirektségű mutatók. Típuskonverziók. Tömbváltozók definiálása. Dinamikus helyfoglalás, a ***malloc()*** és a ***free()*** függvények használata.

Gyakorlat	A gyakorlat tematikája (folytatás)
5.	A felhasználó által definiált adattípusok: a **typedef** és a **struct**. Dinamikus helyfoglalású struktúrák létrehozása és megszüntetése.
6.	Függvények deklarációja és definíciója. Tömb és struktúra típusú paraméterek alkalmazása. Karaktersorozat-argumentumok. Függvényre mutató pointer átadása függvénynek. Függvényszerű makrók készítése.
7.	Listakezelés. Egyirányú és kétirányú láncolt lista kezelése. Új elem beszúrása a láncba. Listaelem törlése a láncból.
8.	Szöveges állományok kezelése. Különféle típusú adatok (karakterek, számok, karaktersorozatok stb.) fájlba írása és állományból való olvasása.
9.	Bináris fájlok kezelése. Bájtonkénti írás/olvasás. Numerikus értékek, tömbök és struktúrák adatainak kiírása bináris állományba, illetve visszaolvasása onnan.
10.	Több modulból álló projektet készítése.
11.	A C nyelv néhány speciális alkalmazása

1. Gyakorlat

Rövid elméleti összefoglaló

- A C nyelv segítségével kisméretű, hatékony, gépfüggetlen programok írhatók.
- **A C nyelv alapelemei:**
  Az ANSI C szabvány a nyelv alapelemeit (nevek, számok, karakterek) *token*nek nevezi. A C nyelv különbözőnek tekinti a kis- és a nagybetűket.
- **A nyelv jelkészlete**: Az angol ábécé kis- és nagybetűi, egyéb látható (! " # % & ' ( ) * + , - /:; < = >? [\ ] ^ _ { | } ~) és nem látható jelek (*whitespace*) (szóköz, vízszintes és függőleges tabulátor, új sor ('\n') stb.)
- **A C nyelv azonosítói**: A legtöbb fordító legfeljebb 32-karakteres neveket használ, az első karakter betű vagy _ (aláhúzásjel), melyet betű, szám, illetve aláhúzásjel követhet.
- **Az ANSI C nyelv kulcsszavait** a legtöbb fordítóprogram kibővíti a szabványos kulcsszavakat saját jelentéssel bíró szavakkal. (Megjegyezzük, hogy ezen bővített kulcsszavak csökkentik a C program hordozhatóságát.)
  Konstansok: C nyelvben: karakteres, egész, felsorolt (ún. **enum**) és lebegőpontos konstansokat használhatunk:
  - A C nyelv egyetlen *mutatókonstanssal* rendelkezik, amelyet a *NULL* szimbólummal jelölünk.
  - *Egész konstansok*: 7, 013, 0x07 (decimális, oktális, hexadecimális),
    16U, 1768L (U, u - **unsigned**, L, l - **long**)
  - *Felsorolt konstansok*: `enum meret{kicsi,kozepes,nagy};`
    A *meret* típusú konstansok értékei: *kicsi* 0, *kozepes* 1, *nagy* 2
  - *Lebegőpontos konstansok*: 213.4, 5.368F, 7.61f (F, f - egyszeres pontosság),
    4.792L (L, l **long double** - nagy pontosság)
    5e-4, 6.2e+9, 2E2, 5E-3 (E, e - exponens, a kitevő).
  - *Sztringkonstansok:* "ez egy szövegkonstans\nkét sorban"
  - *Karakterkonstansok*: '*' egybájtosak.
- **Megjegyzések**: /* */ többsoros megjegyzés.
- **A C nyelv alaptípusai**
  - *Adattípusok*: **char, int, float, double**
  - *Típusmódosítók*: **short, long** – a tárolási hosszat, **singed, unsigned** – az előjel értelmezését szabályozzák.
  - *Típusminősítők*: **const** (csak olvasható), **volatile** (másik futó folyamat is megváltoztathatja).

  Megjegyzés: Nincs külön logikai típus a C nyelvben. Minden nem 0 numerikus érték logikai igazat jelent, a 0 pedig logikai hamisat. A logikai és relációs műveletek eredménye mindig egész (**int**):1, ha az eredmény igaz, és 0, ha az eredmény hamis.
- **Egyszerű változók definiálása**:

  <tárolási osztály><típusmódosítók...> típus változónév <= kezdőérték><,...>;

Például:
```
unsigned int i;                 /* előjel nélküli egész */
long double x = 1.325684e11;    /* nagypontosságú valós */
```

Tárolási osztály: **auto, register, static, extern**;

- A függvényeken belül deklarált változók alaphelyzetben automatikus (**auto**) változók.
- A **register** változót, ha lehetséges, a fordító a számítógép valamely hozzáférhető regiszterébe helyezi (ha nem tudja, akkor a változó a memóriában marad). A **register** változó a program futási sebességét növeli.
- A **static** változót a fordító úgy helyezi el, hogy benne van a kezdőérték, tehát a kezdőértékadás nem igényel futási időt.
- A függvényen kívül definiált változók alapértelmezés szerint külső (más modulból elérhető - **extern**) tárolási osztállyal rendelkeznek. Ha **static**-ként definiáljuk a függvényeket vagy a függvényen kívüli változókat, akkor azok láthatóságát az adott forrásmodulban korlátozzuk.

- **Operátorok és kifejezések**:
  - *Aritmetikai operátorok*: / , % , * , + , – (|osztás, maradék, szorzás|, |összeadás, kivonás|. Az operátorok felsorolása a műveletek precedenciája szerint történt, az azonos szintű precedenciákat a | vonal fogja össze.)
  - *Precedencia-szabály:* Műveletek precedenciája, zárójelezés.
  - *Értékadó utasítás* : jobbérték és a balérték fogalma.
    a = 4.3; a = a + 2.1; a + = 2.1; de lehet – =, *=, /=, %=
  - *Csoportosítási szabály* : i = j = k = 0; (jobbról-balra szabály)

- **A C program szerkezete**

  A C nyelvű program egy vagy több forrásfájlban helyezkedik el, melyek kiterjesztése .C. A forrásprogramhoz az ún. deklarációs (include, header, fej-) állományok csatlakoznak (.H), melyeket a *#include* előfordító utasítás segítségével építünk be a forrásállományba.

  - A C programhoz egy ***main()*** függvény tartozik, amely kijelöli a program belépési pontját – azaz a program futása a ***main()*** függvény indításával kezdődik.
  - A be- (***scanf***) és kimenő (***printf***) adatfolyam kezelő függvények deklarációját (prototípusát) az *STDIO.H* állomány tartalmazza, melyet a programba a *#include* előfordító utasítással építünk be:

    ```
    #include <stdio.h>
    ```

  - A ***printf()*** függvény adott formátum szerint jelenít meg adatokat. A formátumot az ún. formátumsztring határozza meg. A formátumsztringet kettős idézőjelek ("") között kell megadni. A benne megadott szöveg a szabványos kimenetre (képernyőre) íródik ki. A kiíratásnál új sort a '\n' karakterrel emelhetünk.

    printf("formátumsztring", argumentumlista);

    A változók kiíratása a típusokra vonatkozó konverzióval történik:

    ```
    int %d, long  %ld, float %f, double %lf, char %c, sztring %s
    ```

  - Adatok beolvasása: a ***scanf()*** függvénnyel végezhető. A változók beolvasására a típusnak megfelelő konverziót használjuk a formátumsztringben. Az adatnak a változóba történő beírásához a változó címét kell megadni.
    A változó címét a változó neve előtti & operátor képezi.

A *scanf()* függvény általános formája:

scanf("formátumsztring", argumentumlista);

Megjegyzés: Karaktersorozatot tároló változónak a ***scanf()*** mellett az ***strcpy()*** (*sztring copy*) függvénnyel adhatunk értéket! A sztringkezelő függvények prototípusai a *STRING.H* fejállományban találhatók.

```
/*MINTA1_01*/
#include <stdio.h>
#include <string.h>
int main()
{ /* konstansok */
  const double pi = 3.141592653589;
  const char jel = '!';
  /* deklaráció */
  int i,j;        /* egész */
  long k;         /* hosszú egész*/
  float a,b;      /* valós */
  double x,y;     /* duplapontos valós */
  char c;         /* karakter */
  char sz[11];    /* 10 karakteres sztring */
  /* utasítások */
  i = 13; j = -45;
  k = 1234556;
  a = -2.5678; b = 123.98765;
  x = 423.67; y = 1.5e-4;
  c = '*';
  strcpy(sz, "C nyelv");
  printf("pi erteke: %lf\n",pi);
  printf("i = %d\nj = %d\n", i,j);
  printf("k = %ld\n", k);
  printf("x = %lf\ny = %lf\n",x, y);
  printf("c = %c\n",c);
  printf("sz = %s\n",sz);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
pi erteke: 3.141593
i = 13
j = -45
k = 1234556
x = 423.670000
y = 0.000150
c = *
sz = C nyelv
```

Megjegyzés: A C nyelvben két egész szám hányadosa egész, azaz egész osztás történik. A **double** típusú bal oldal ellenére is értékvesztés (az eredmény egésszé csonkolása) áll fenn. Használnunk kell a típusmódosítást (*cast*)! Ha van lehetőség rá, akkor kétféle módon is történhet az átalakítás:

- A számlálónak, vagy a nevezőnek a típusát alakítjuk át valóssá (**double**):

```
oszt2 = (double)i /j;
oszt2 = i /(double)j;
```

- Ha például a nevező konstans, valós számmá alakíthatjuk a tizedesponttal, az eredmény akkor is valós lesz:

```
oszt3 = i / 5.0;
```

A *MINTA1_02* feladat bemutatja az eredmények kiírásánál a mezőszélesség használatát, illetve a tizedes jegyek számának beállítását.

A mezőszélesség előjele is meghatározó:

- Pozitív szám esetében az eredmény a megadott mezőszélességben jobbra igazítva jelenik meg.
- Negatív szám esetében az eredmény a megadott mezőszélességben balra igazítva látható.

Az egész számok megjelenítése:

- *%wd,* ahol a *w* értéke adja a mezőszélességet. A példában az *ered1* tartalma jobbra igazítva jelenik meg. Ha a szám kisebb a mezőszélességnél, akkor szóköz karakter tölti ki a nem használt pozíciókat:

```
printf("ered1: %3d\n",ered1);
```

- A *w* értéke lehet negatív, ebben az esetben az eredmény balra igazítva lesz látható a kijelölt szélességű mezőn belül.

```
printf("ered2: %-3d\n",ered2);
```

Valós számok esetében is adhatunk mezőszélességet, melybe az összes karakter (az előjel, az egészek száma, a tizedespont, tizedes jegyek száma is) beleszámít, majd ponttal a tizedes jegyek számát adjuk meg. Itt is lehet negatív mezőszélességet használni.

```
printf("b: %7.3lf\n",b);
```

```
/*MINTA1_02*/
#include <stdio.h>
int main()
{
  int i = 2, j = 5, ered1, ered2;
  double a, b, oszt1, oszt2, oszt3, szor;
  ered1 = i * j -1;
  ered2 = i * (j - 1);
  a = i + j;
  b = i - j + 0.5;
  oszt1 = i / j;
  oszt2 = (double)i /j;
  oszt3 = i / 5.0;
  szor = oszt2 * -100.566;
  printf("ered1: %3d\n",ered1);
  printf("ered2: %-3d\n",ered2);
  printf("a: %lf\n",a);
  printf("b: %7.3lf\n",b);
  printf("oszt1: %lf\n",oszt1);
  printf("oszt2: %8.2lf\n",oszt2);
  printf("oszt2: %8.3lf\n",oszt3);
  printf("szorzat: %e\n",szor);
  printf("szorzat: %10.2e\n",szor);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
ered1:   9
ered2: 8
a: 7.000000
b:  -2.500
oszt1: 0.000000
oszt2:     0.40
oszt2:    0.400
szor: -4.022640e+001
szor: -4.02e+001
```

Olvassuk be a téglalap két oldalának hosszát! Írjunk programot, amely kiszámítja a téglalap területét és kerületét!
Az adatok beolvasása a következő módon történik:

1. Először rákérdezünk az adatra soremelés nélkül:
   ```
   printf("a oldal : ");
   ```
2. Beolvassuk az adatot, megadva a változó címét (&), ahol az adatot tárolni szeretnénk:
   ```
   scanf("%lf",&a);
   ```
   Megjegyzés: a beolvasásnál ne használjunk soremelést ('\n'), mivel a ***scanf()*** nem olvassa be a tagoló karaktereket (szóközt és az újsort ('\n')).

```
/*MINTA1_03*/
#include <stdio.h>
int main()
{
  double a, b, ter, ker;
  printf("a oldal : "); scanf("%lf",&a); /* adat olvasása */
  printf("b oldal : "); scanf("%lf",&b); /* adat olvasása */
  ker = (a + b)/2; /* a téglalap kerületének számítása */
  ter = (a * b);   /* a téglalap területének számítása */
  /* az eredmények kiíratása */
  printf("Teglalap kerulete: %8.2lf\n", ker);
  printf("Teglalap terulete: %8.2lf\n", ter); */
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
a oldal : 2
b oldal : 3
Teglalap kerulete:     2.50
Teglalap terulete:     6.00
```

A feladatok megoldásánál gyakran használunk matematikai függvényeket, melyek hívásához *MATH.H* fejállományt kell beszerkesztenünk a programba:

```
#include <math.h>
```

gyökvonás	: x gyöke	`y = sqrt(x);` vagy `y = pow(x,1./2);`
koszinusz	: x koszinusza	`y = cos(x);`
szinusz	: x szinusza	`y = sin(x);`
hatványozás	: a^n	`y = pow(a,n);` vagy `y = exp(log(x)*n));`
exp függvény	: e^x	`y = exp(x);`
természetes log.	: ln x	`y = log(x);`

A *MINTA1_04* mintafeladat néhány matematikai függvény használatát mutatja be:

```
/*MINTA1_04*/
#include <stdio.h>
#include <math.h>
#define PI 3.141592654  /* PI konstansként való megadása */
```

```
int main()
{
  double a = 4, b = 2, kobe, gyok1, gyok2;
  double szog = 30, fokrad, ered;
  kobe = pow(b, 3);      /* köbre emelés */
  gyok1 = sqrt(a);       /* gyökvonás */
  gyok2 = pow(a, 1.0/2); /* gyökvonás */
  fokrad = szog*PI/180; /* fok radiánba való átalakítása */
  ered = sin(fokrad);    /* sin függvény paramétere radián */
  printf("%.3lf kobe : %.3lf\n",b, kobe);
  printf("%.3lf gyoke: %.3lf\n",a, gyok1);
  printf("%.3lf gyoke: %.3lf\n",a, gyok2);
  printf("\nfok: %.4lf radian: %.4lf\n",szog, fokrad);
  printf("%.4lf fok szinusza: %.4lf\n",szog, ered);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
2.000 kobe : 8.000
4.000 gyoke: 2.000
4.000 gyoke: 2.000

fok: 30.0000 radian: 0.5236
30.0000 fok szinusza: 0.5000
```

Feladatok

1. Olvassuk be egy téglatest három oldalának hosszát! Írjunk programot, amely kiszámítja a téglatest térfogatát és felszínét! (GYAK1_1)

   ```
   double a,b,c, felsz, terf;
   ```

   A program futásának eredménye:

   ```
   A oldal: 1
   B oldal: 2
   C oldal: 3
   Felszin :    22.00
   Terfogat:     6.00
   ```

2. Olvassunk be egy valós számot (*a*), valamint egy egész kitevőt (*kitevo*)! Számítsuk ki a valós számnak a hatványát és a gyökét! Használjuk a matematikai függvényeket! (GYAK1_2)

   ```
   double a, hatvany, gyok;
   int kitevo;
   ```

3. Írjunk programot, amely beolvassa a derékszögű háromszög átfogóját (*atfogo*) és az egyik hegyes szögét (*alfa*). Számítsuk ki a háromszög két befogóját! (GYAK1_3)

   ```
   double befogo1, befogo2, atfogo, alfa;
   ```

   $a = c \cdot \sin\alpha \qquad b = c \cdot \cos\alpha$

4. Olvassuk be a sugár hosszát! Írjunk programot, amely kiszámítja a kör kerületét, területét, a gömb felszínét és térfogatát! (GYAK1_4)

```
double r, korker, korter, gombfelsz, gombterf;
```

$$körkerület = 2 \cdot r \cdot \pi \qquad körterület = r^2 \cdot \pi$$

$$gömbfelszín = 4 \cdot r^2 \cdot \pi \qquad gömbtérfogat = \frac{4r^3 \cdot \pi}{3}$$

A program futásának eredménye:

```
sugar: 1
kor kerulete  : 6.283185
kor terulete  : 3.141593
gomb felszine : 12.566371
gomb terfogata: 4.188790
```

5. Olvassuk be egy kocka élének hosszát! Írjunk programot, amely kiszámítja a kocka felszínét, térfogatát, testátlóját és lapátlóját! (GYAK1_5)

```
double a, lapatlo, testatlo, felsz, terf;
```

$$lapátló = a \cdot \sqrt{2} \qquad testátló = a \cdot \sqrt{3}$$

$$felszíne = 6 \cdot a^2 \qquad térfogata = a^3$$

A program futásának eredménye:

```
kocka oldalele: 1
lapatlo   : 1.414214
testatlo  : 1.732051
kocka felszine : 6.000000
kocka terfogata: 1.000000
```

6. Írjunk programot, amely beolvassa egy háromszög két oldalát és a közbezárt szöget! Számítsuk ki a harmadik oldalt! (GYAK1_6)

```
double a, b, c, gamma;
```

$$c = \sqrt{a^2 + b^2 - 2 \cdot a \cdot b \cdot \cos\gamma}$$

A program futásának eredménye:

```
a oldal: 3
b oldal: 4
gamma  : 30
c oldal:     2.05
```

7. Olvassuk be a derékszögű háromszög két befogóját, és számítsuk ki az átfogót, a háromszög területét és kerületét! (GYAK1_7)

```
double befogo1, befogo2, atfogo, ter, ker;
```

A program futásának eredménye:

```
1. befogo: 3
2. befogo: 4
atfogo:     5.00
terulet:     6.00
kerulet:    12.00
```

2. Gyakorlat

Rövid elméleti összefoglaló

- **Léptetető operátorok**
  - ++ (inkrement), a változó értékének eggyel való növelése: i++; ++i;
  - -- (dekrement), a változó értékének eggyel való csökkentése: j--; --j;

Megjegyzés: A prefixes alak (++i, --j) használata esetén a léptetés a kifejezés kiértékelése előtt megy végbe, és a változó az új értékkel vesz részt a kifejezés kiértékelésében. A postfixes alak (i++, j--) használata esetén a léptetés a kifejezés kiértékelése után történik meg, mellékhatásként.

```
/*MINTA2_01*/
#include <stdio.h>
int main()
{
  int a, b, sum;
  char *format=" a= %d  b= %d  sum= %d\n";
  a = b = 5;
  sum = a + b;     printf(format,a,b,sum);
  sum = a++ + b;   printf(format,a,b,sum);
  sum = ++a + b;   printf(format,a,b,sum);
  sum = a + b;
  printf(" sum= %d  a= %d  b=%d \n",sum,a,b);
  sum = --a + b;   printf(format,a,b,sum);
  sum = a-- + b;   printf(format,a,b,sum);
  sum = a + b;     printf(format,a,b,sum);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
a= 5  b= 5  sum= 10
a= 6  b= 5  sum= 10
a= 7  b= 5  sum= 12
sum= 12  a= 7  b=5
a= 6  b= 5  sum= 11
a= 5  b= 5  sum= 11
a= 5  b= 5  sum= 10
```

Mit látunk a képernyőn?

```
/*MINTA2_02*/
#include <stdio.h>
int main()
{
  int a = 8, b = -12, c, d;
  printf("Mit latunk a kepernyon\n");
  a++;   c = -a++;
  b++;   d = - ++b;
  printf(" a = %d  b = %d  c = %d  d = %d \n", a, b, c, d );
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
Mit latunk a kepernyon
 a = 10  b = -10  c = -9  d = 10
```

- **Bitműveletek** (a felsorolás a precedencia sorrendjében történt):
  - ~ 1-es komplemens (bitenkénti tagadó),
  - & bitenkénti ÉS,
  - | bitenkénti VAGY,
  - ^ bitenkénti kizáró VAGY.

A *MINTA2_03* mintafeladat bemutatja a bitműveletek használatát.

```
/*MINTA2_03*/
#include <stdio.h>
int main()
{
  int a,b,c;
  char *format1 = "%04x  %s  %04x = %04x\n";
  char *format2 = "%c %04x = %04x\n";

  a = 0x0ff0; b = 0xff00;
  printf(" a= %d  %04x  b= %d  %04x\n",a,a,b,b);

  c=a&b;  printf(format1,a," &",b,c);
  c=a|b;  printf(format1,a," |",b,c);
  c=a^b;  printf(format1,a," ^",b,c);
  c=~a;   printf(format2,'~',a,c);
  c=-a;   printf(format2,'-',a,c);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
a= 4080  0ff0  b= 65280  ff00
0ff0   &  ff00 = 0f00
0ff0   |  ff00 = fff0
0ff0   ^  ff00 = f0f0
~ 0ff0 = fffff00f
- 0ff0 = fffff010
```

- **Biteltoló műveletek**
  - << eltolás balra a << n (2^n értékkel való szorzás):
    `szorzas = b << 1; /* kettővel való szorzás */`
  - >> eltolás jobbra a >>m (2^m értékkel való osztás)
    `osztas = c >> 1; /* kettővel való osztás */`

A *MINTA2_04* mintafeladat a biteltoló műveleteket mutatja be.

```
/*MINTA2_04*/
#include <stdio.h>
int main()
{
  int a, b = 4, c;
  char *format1 = "%04x  %s  %04x = %04x %d\n";
  char *format2 = "%c %04x = %04x %d\n";
```

```
  a = 0x0012;
  printf("a= %d  %04x\n",a, a);
  c = a << b; printf(format1,a,"<<",b,c,c);
  c = a >> b; printf(format1,a,">>",b,c,c);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
a= 18  0012
0012  <<  0004 = 0120 288
0012  >>  0004 = 0001 1
```

Az *a* változó értékét tízes számrendszerben és hexadecimális számrendszerben is kiíratjuk.

Az eredmény második sorában:
Léptetés előtt: `0000 0000 0001 0010`
Léptetés után: `0000 0001 0010 0000`
`  hexadecimalis 120 -> tizes számrendszerben 288`

Az eredmény harmadik sorában:
Léptetés előtt: `0000 0000 0001 0010`
Léptetés után: `0000 0000 0000 0001`
`  hexadecimalis 1 -> tizes számrendszerben 1`

Megjegyzés: A kiléptetett bitek elvesznek!

- A **sizeof** operátor megadja tetszőleges változó, típus, kifejezés típusának megfelelő bájtméretet.
  - **sizeof** változó
  - **sizeof** (típusnév)

```
/*MINTA2_05*/
#include <stdio.h>
int main()
{
  int j = 10, jhossza, thossza,
      wthossza, whossza;
  float t = 2.5;
  double w = 12.56;
  jhossza = sizeof j;
  thossza = sizeof t;
  wthossza = sizeof(double);
  whossza = sizeof w;
  printf("int merete: %d \n",jhossza);
  printf("float merete: %d \n",thossza);
  printf("double merete: %d \n",wthossza);
  printf("double merete: %d \n",whossza);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
int merete: 4
float merete: 4
double merete: 8
double merete: 8
```

- **Összehasonlító és logikai operátorok**
  - Relációjelek: <, <=, >, >=, == (egyenlő), != (nem egyenlő)
  - Logikai operátorok: && (ÉS), || (VAGY), ! (NEM)

A relációs és logikai műveletek eredménye mindig logikai érték, ahol az igazat 1, a hamisat 0 reprezentálja. A C nyelvben minden nem 0 numerikus érték logikai értékelése igaz, minden numerikus 0 érték logikai hamisat jelent.

A relációjelek és a logikai operátorok használatára az **if** utasításnál látunk mintafeladatokat.

- **A C++ nyelv utasításai**
  - Tetszőleges kifejezés utasítás lesz, ha pontosvesszőt (;) helyezünk mögé.
  - A kapcsos zárójelek { } a logikailag összefüggő definíciókat, deklarációkat és utasításokat egyetlen összetett utasításba, blokkba fogják össze.
  - Feltételes operátor három operandussal rendelkezik:

```
feltétel ? igaz_kif : hamis_kif;
/* a nagyobb adatot választja ki: */
max = x > y ? x : y;
/* szintén a nagyobb adatot választja ki: */
x > y ? max = x : max = y;
```

  - **if** (feltételes) utasítás:

if (*kifejezés*)
{
utasítások;
}
	Ha a *kifejezés* igaz (értéke nem nulla), akkor a { } zárójelben lévő *utasításokat* hajtja végre.

if (*kifejezés*) *utasítás1*; **else** *utasitas2*;	Ha a *kifejezés* igaz (értéke nem nulla), akkor az *utasítás1*-et hajtja végre. Ha a *kifejezés* nem igaz (értéke nulla), akkor az *utasítás2*-őt dolgozza fel.

A feltételes operátor tehát egy olyan "**if**", aminek "*értéke van*".

Három valós szám közül válasszuk ki a legnagyobbat!

```
/*MINTA2_06*/
#include <stdio.h>
int main()
{ double a,b,c,max;
  printf(" a : "); scanf("%lf",&a);
  printf(" b : "); scanf("%lf",&b);
  printf(" c : "); scanf("%lf",&c);
  max = c;
  if ( a > b )
  {
    if( a > c ) max = a;
  }
  else
  {
    if (b > c ) max = b;
  }
```

A program futásának eredménye:

```
 a : 3
 b : 5
 c : 1
Maximum: 5.000000
```

```
  printf("Maximum: %lf\n",max);
  fflush(stdin);getchar();
  return 0;
}
```

Módosítsuk a *MINTA2_06* mintafeladatot, használjunk logikai operátorokat!

```
/*MINTA2_07*/
#include <stdio.h>
int main()
{
  double a,b,c,max;
  printf(" a : "); scanf("%lf",&a);
  printf(" b : "); scanf("%lf",&b);
  printf(" c : "); scanf("%lf",&c);
  max = c;
  if ( a > b && a > c ) max = a;
  else if (b > c ) max = b;
  printf("Maximum: %lf\n",max);
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
 a : 2
 b : 4
 c : 1
Maximum: 4.000000
```

Megjegyzés: A ***getchar()*** függvénnyel egy karaktert tudunk beolvasni. A *MINTA2_08* mintafeladatban a beolvasott karakterről döntsük el, hogy szám, kisbetű vagy nagybetű volt! Egyéb karakter esetén nincs kijelzés.

```
/*MINTA2_08*/
#include <stdio.h>
int main()
{
 int ch;
 printf("karakter: ");
 ch = getchar();
 if (ch >= '0' && ch <= '9')
   printf("szam      : %d %c\n",ch,ch);
 if (ch >= 'A' && ch <= 'Z')
   printf("nagy betu: %d %c\n",ch,ch);
 if (ch >= 'a' && ch <= 'z')
   printf("kis betu : %d %c\n",ch,ch);
 fflush(stdin);getchar();
 return 0;
}
```

A program futásainak eredménye:

```
karakter: a
kis betu : 97 a

karakter: B
nagy betu: 66 B

karakter: 3
szam     : 51 3
```

Alkalmazzunk függvényeket a számjegy, kis- és nagybetű vizsgálatára. A függvények, (makrók) használatára a *CTYPE.H* fejállományt kell a programba beszerkeszteni.

- ***isdigit(ch)*** akkor ad igaz értéket, ha a *ch* argumentum számjegy,
- ***islower(ch)*** akkor ad igaz értéket, ha a *ch* argumentum kisbetű,
- ***isupper(ch)*** akkor ad igaz értéket, ha a *ch* argumentum nagybetű.

```c
/*MINTA2_09*/
#include <stdio.h>
#include <ctype.h>
int main()
{
 int ch;
 printf("karakter: ");
 ch = getchar();
 if (isdigit(ch)) /* számjegy */
    printf("szam      : %d %c\n",ch,ch);
 else
 if (islower(ch)) /* kisbetű */
    printf("kis betu : %d %c\n",ch,ch);
 else
 if (isupper(ch)) /* nagybetű* /
    printf("nagy betu: %d %c\n",ch,ch);
 else printf("egyik sem!\n");
 fflush(stdin);getchar();
 return 0;
}
```

A program futásainak eredménye:

```
karakter: 3
szam      : 51 3

karakter: c
kis betu : 99 c

karakter: A
nagy betu: 65 A
```

– A **switch** utasítás (többirányú elágaztatás, a *kifejezés* csak sorszámozott típusú lehet).

switch (*kifejezés*)
{
  case *konst_kifejezés*:
    utasítások; **break**;
    ...
  default:
    utasitások;
}

– A **break** utasítással kiléphetünk a **switch**-et alkotó blokkból.

Az egész számként beolvasott osztályzatot (1-5) írjuk vissza szövegesen a **switch** utasítás használatával!

```c
/*MINTA2_10*/
#include <stdio.h>
int main()
{
  int osztalyzat;
  printf("Osztalyzat (1-5): ");
  scanf("%d",&osztalyzat);
  switch (osztalyzat)
  {
    case 1: printf("Elegtelen\n"); break;
    case 2: printf("Elegseges\n"); break;
    case 3: printf("Kozepes\n"); break;
    case 4: printf("Jo\n"); break;
    case 5: printf("Jeles\n"); break;
    default:
      printf("Hibas osztalyzat!\n");
  }
```

A program futásainak eredménye:

```
Osztalyzat (1-5): 5
Jeles

Osztalyzat (1-5): 6
Hibas osztalyzat!
```

```
   fflush(stdin);getchar();
   return 0;
}
```

Feladatok

1. Olvassunk be két számot, és a nagyobbat jelenítsük meg! (GYAK2_1)

```
double adat1, adat2, nagyobb;
```

2. Olvassunk be három pozitív számot és számítsuk ki az összegüket, átlagukat és mértani közepüket! (GYAK2_2)

```
double adat1, adat2, adat3, osszeg, atlag, mkozep;
```

A program futásának eredménye:

```
1. adat: 1
2. adat: 2
3. adat: 3
osszeg : 6.000000
atlag  : 2.000000
mertani kozep: 1.817121
```

3. Olvassunk be egy valós számot, írjuk vissza, hogy a beolvasott szám pozitív, negatív, vagy zérus! (GYAK2_3)

A program futásának eredménye:

```
1. adat: 3
szam pozitiv
```

4. Olvassunk be egy egész pozitív számot, ha páratlan emeljük négyzetre, ha páros vonjunk belőle négyzetgyököt! Írjuk ki az eredményt! (GYAK2_4)

A program futásainak eredménye:

```
adat: 2                    adat: 3
paros                      paratlan
negyzete:      4.00        gyoke:      1.73
```

5. Írjunk programot, amely beolvassa a háromszög három oldalát, és ellenőrzi, hogy a megadott három oldalból megszerkeszthető-e a háromszög! Ha megszerkeszthető, akkor kiszámítja a háromszög kerületét és a területét a *Heron* képlet felhasználásával, különben hibajelzést ad! (GYAK2_5)
Használjuk az alábbi definíciókat:

```
double a, b, c, ter, ker;
```

$kerület = a + b + c$ $\quad s = \frac{a+b+c}{2}$ a fél kerület

$terület = \sqrt{s \cdot (s-a) \cdot (s-b) \cdot (s-c)}$

A háromszög akkor szerkeszthető meg, ha bármely két oldal hosszának összege nagyobb a harmadik oldal hosszánál mindhárom oldalra nézve:

```
if (a + b > c && b + c > a && a + c > b)
```

A program futásának eredménye:

```
a oldal: 3
b oldal: 4
c oldal: 5
kerulet:    12.00
terulet:     6.00
```

6. Írjunk programot, amely Celsiusból Fahrenheit fokra vált! Kérdezzünk rá: *"Váltás (c/f): "*, *c* vagy *C* karakter leütése esetén Celsius fokot olvas és Fahrenheit fokra vált, f illetve F leütésekor pedig Fahrenheit fokot olvas és Celsius fokra vált. (GYAK2_6)

A program futásainak eredménye:

```
Valtas (c/f): c          Valtas (c/f): f
Celsius : 0              Fahrenheit : 32
Fahrenheit:    32.00     Celsius:     0.00
```

7. Olvassunk be három számot az *a*, *b*, *c* változókba! Rendezzük sorrendbe, hogy az *a*, *b*, *c* a növekvő sorrendet tartalmazza! (GYAK2_7)

```
int a, b, c, x;
```

A program futásának eredménye:

```
1. adat: 5
2. adat: 1
3. adat: 3

Novekvo sorrendben
1. adat: 1
2. adat: 3
3. adat: 5
```

8. Olvassunk be három számot és válasszuk ki a legkisebbet! (GYAK2_8)

```
int a, b, c, min;
```

A program futásának eredménye:

```
1. adat: 5
2. adat: 1
3. adat: 3

Legkisebb adat: 1
```

3. Gyakorlat

Rövid elméleti összefoglaló

- **Ciklusutasítások**
  - **while** ciklus: a ciklusmag mindaddig végrehajtódik, míg a kifejezés értéke igaz (nem nulla)
    while (*kifejezés*)
    {
    utasítás(ok);
    }

```
/*MINTA3_01*/
#include <stdio.h>
int main()
{
  int i, paros = 0, oszt3 = 0;
  i = 1;
  while (i<15)
  {  if (i % 2 == 0) paros++;
     if (!(i % 3)) oszt3++;
     i++;
  }
  printf("Parosak szama: %d\n",paros);
  printf("3-mal oszthatok szama: %d\n",
         oszt3);
  getchar();
  return 0;
}
```

Számláljuk meg a páros számokat és a 3-mal oszthatókat 1 és 14 között **while** ciklussal!
A **while** ciklusnál a ciklusváltozót a ciklus utasítás előtt az adott kezdőértékre állítjuk be (i = 1). Úgy adjuk meg a feltételt (i<15), hogy a kezdőérték mellett és a kívánt végértékig igaz legyen, mert csak akkor fogja a ciklus törzsében lévő utasításokat végrehajtani. A ciklus belsejében a ciklusváltozó értékét változtatnunk kell (i++), oly módon, hogy a ciklus elérje a végértékét, különben végtelen ciklust kapunk. A ciklus akkor áll le, amikor a feltétel hamis lesz.

A program futásának eredménye:

```
Parosak szama: 7
3-mal oszthatok szama: 4
```

  - **for** ciklus: a ciklusmagban az utasítás a megadott számsorozat mentén kerül végrehajtásra.
    for (*kezdőérték_kif; feltétel_kif; léptető_kif*)
    utasítás;

    Ha több kezdőértéket adunk a ciklus fejében, vesszővel kell azokat elválasztanunk.

```
/*MINTA3_02*/
#include <stdio.h>
int main()
{
  int i, paros = 0, oszt3 = 0;
  for(i = 1,paros = 0,oszt3 = 0; i<15; i++)
  {
     if (i % 2 == 0) paros++;
     if (!(i % 3)) oszt3++;
  }
  printf("Parosak szama: %d\n",paros);
```

Számláljuk meg a páros számokat és a 3-mal oszthatókat 1 és 14 között **for** ciklussal!

A program futásának eredménye:

```
Parosak szama: 7
3-mal oszthatok szama: 4
```

```
  printf("3-mal oszthatok szama: %d\n",
         oszt3);
  getchar();
  return 0;
}
```

– **do - while** ciklus : legalább egyszer végrehajtja az utasítást, majd a továbbiakban addig addig ismétli az utasítást, míg a kifejezés értéke igaz (nem nulla)

do
{
utasítás;
}**while** (*kifejezés*);

```
/*MINTA3_03*/
#include <stdio.h>
int main()
{
  int i, paros = 0, oszt3 = 0;
  i = 1;
  do
  {
     if (i % 2 == 0) paros++;
     if (!(i % 3)) oszt3++;
     i++;
  }while (i<15);
  printf("Parosak szama: %d\n",paros);
  printf("3-mal oszthatok szama: %d\n",
         oszt3);
  getchar();
  return 0;
}
```

Számláljuk meg a páros számokat és a 3-mal oszthatókat 1 és 14 között **do-while** ciklussal!
Beállítjuk a ciklusváltozó kezdőértékét:
i = 1;

A ciklus törzse egyszer feltétel nélkül lefut, mivel a feltételt a ciklus végén vizsgáljuk. Ugyanakkor a ciklusváltozó értékét (i++) változtatnunk kell, különben végtelen ciklust kapunk. A ciklus a feltétel hamisra válásakor áll le.

A program futásának eredménye:

```
Parosak szama: 7
3-mal oszthatok szama: 4
```

- A **continue** utasítás
  A **continue** utasítás a ciklusok soron következő iterációját indítja a mögötte álló utasítások átlépésével.
- A **break** utasítással kiléphetünk a ciklusból (blokkból).
- A **goto** utasítással a megadott címkéjű sorra adjuk a vezérlést:
  goto *címke*:
  utasítások
  címke:

Feladat: Szorozzuk össze a ciklus változójának értékét, és számláljuk meg az adatokat! A ciklusváltozó 2-es és a 4-es értékénél a **continue** kihagyja az iterációt, és a következőnél folytatja. A ciklusból a 7. lépésnél lépünk ki a **break** utasítással. A szorzat és a darabszám ismeretében kiszámítjuk a mértani közepet logaritmus használatával, illetve a *pow()* függvénnyel.
(*MINTA3_4*)

```
/*MINTA3_04*/
#include <stdio.h>
#include <math.h>
int main()
{
  double mkozep1, mkozep2, szorzat;
  int i, db;
  for (i = 1, szorzat = 1, db = 0; i<9; i++)
  {
    if (i == 7) break;
    if (i == 2 || i == 4) continue;
    szorzat *= i;
    db++;
    printf("i: %d  szorzat: %8.2lf  db: %d\n",i,szorzat, db);
  }
  mkozep1 = exp(log(szorzat)/db);
  mkozep2 = pow(szorzat, 1.0/db);
  printf("Mertani kozep: %lf\n", mkozep1);
  printf("Mertani kozep: %lf\n", mkozep2);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
i: 1  szorzat:     1.00  db: 1
i: 3  szorzat:     3.00  db: 2
i: 5  szorzat:    15.00  db: 3
i: 6  szorzat:    90.00  db: 4
Mertani kozep: 3.080070
Mertani kozep: 3.080070
```

Feladat: A ***getchar()*** függvénnyel olvassunk karaktereket, és számláljuk meg, hogy hány új sort, vízszintes *tab* és szóközt karaktert ütöttünk le! A kilépés a pont (.) karakterrel történjen!

```
/*MINTA3_05*/
#include <stdio.h>
int main()
{
  int ch, n, n1;
  n = n1 = 0;
  printf("Karakterek olvasas: (vege: .): \n");
  while ((ch = getchar()) != '.')
  {
     n++;
     if (ch == '\n' || ch == '\t' || ch == ' ') n1++;
  }
  printf("\nstdin-rol olvasott osszes karakter szama: %d\n",n);
  printf("uj sorok, tabulatorok es szokozok szama : %d\n",n1);
  fflush(stdin); getchar();
  return 0;
}
```

A program futásának eredménye:

```
Karakterek olvasas: (vege: .):
a b     c       d
d
e.

stdin-rol olvasott osszes karakter szama: 11
uj sorok, tabulatorok es szokozok szama : 5
```

Feladatok

1. Olvassuk be az adatok darabszámát egy *db* változóba! A **while** ciklus használatával *db* darab egész számot olvasva, határozzuk meg a páratlan és az öttel osztható adatok számát! (GYAK3_1)

```
int i, db, paratlan = 0, oszt5 = 0;
int adat;
```

A program futásának eredménye:

```
adatok szama: 5
1
2
3
4
5

paratlanok szama: 3
5-tel oszthatok szama: 1
```

2. Olvassunk be egy kijelentő mondatot karakterenként a pontig a **while** ciklus használatával. Állapítsuk meg mennyi a beolvasott szöveg hossza, valamint mennyi *a* karakter és szóköz (*space*) karakter van a szövegben! (GYAK3_2)

```
int ch, adb = 0, spdb = 0, osszdb = 0;
while ((ch = getchar()) != '.')
{
  . . .
}
```

A program futásának eredménye:

```
Szoveg beolvasasa a pontig (.):
Szep a pettyes labda.
osszes karakterek szama: 20
a karakterek szama: 3
ures helyek szama : 3
```

3. Olvassunk be egy pozitív egész számot 2 - 50 között! A helyes beolvasás ellenőrzéséhez használjunk **do - while** ciklust! Írjuk ki a beolvasott szám osztóit **while** ciklus segítségével! (GYAK3_3)

```
int szam, i;
```

A program futásainak eredménye:

```
Pozitiv egesz szam (2 es 50 kozott): 21
Osztoi:
3
7

Pozitiv egesz szam (2 es 50 kozott): 3
Osztoi:
Nincs osztoja, primszam!
```

4. Írjunk programot kamatos kamat számítására! Az *alap* változóba olvassuk be a kamatoztatni kívánt összeget, a *kamat* változóba a kamatlábat és az *ev* változóba a lekötés időtartamát! (GYAK3_4)

```
double alap, toke;
int ev, i, kamat;

. . .
toke = alap;
for (i = 1; i<=ev; i++)
{
  toke += toke * kamat/100;
}
. . .
```

A program futásának eredménye:

```
alap : 100
kamat: 10
ev   : 10
Kamatos kamattal novelt toke:   259.37
```

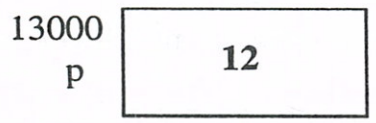
4. Gyakorlat

Rövid elméleti összefoglaló

- **Mutató:**

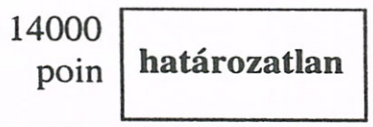
`int p = 12, x;`	
`int *poin;`	a *poin* **int**-re mutató pointer (címet) tárol.
`poin = &p;`	a *poin* által megcímzett (mutató) adat és a *p* egy és ugyanaz.
`x = *poin;`	*x* értéke 12 lesz.

Bővebb magyarázat:
Az `int p =12;` definíció fordításakor a fordító elhelyezi a *p* változót pl. a 13000 memóriacímre, amely mögött 4 bájt szabad memóriaterület van egy egész szám tárolására. Mivel a változó már a definíció során kapott kezdőértéket, a 13000 memóriacímen található tároló tartalma a 12 egész szám lesz.



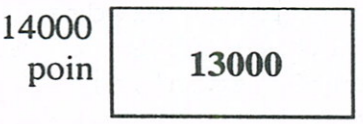
Hozzáférhetünk a *p* címéhez – a *&p* kifejezés a *p* címét jelenti –, a példánkban ez 13000.

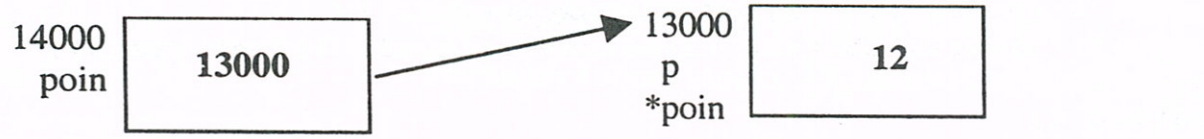
Az `int *poin;` definíció fordításakor a fordító elhelyezi a *poin* változót pl. a 14000 memóriacímre, amely mögött 4 bájt szabad memóriaterület van, hogy egy memóriacím elférjen benne. Ebben az esetben a *poin* egy **int** típusra mutató címet fogad el a továbbiakban, azonban kezdetben csak egy véletlenszerű értéket tartalmaz. Hibázik a program, ha nem gondoskodunk a megfelelő értékadásról.



`poin = &p;` utasítás végrehajtásának hatására

a *poin* felveszi a *p* változó címét:





címet tartalmaz

a *poin azt az **int** típusú tárolót jelöli, ahová a *poin* mutat

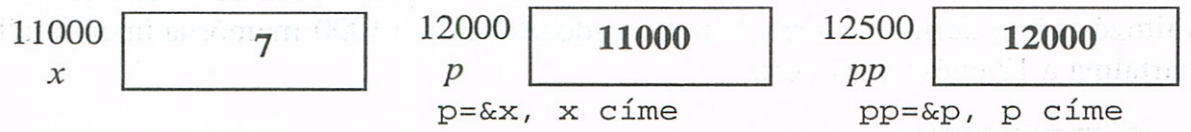
Összefoglalva: a *poin* változó ebben a példában **int**-re, azaz egész változóra mutató címet tárol, a **poin* kifejezés a megcímzett adatot (itt a *p* változó tartalmát) jelenti (jelen példában a 12-őt).

- **void* mutató használata**

A **void*** a C nyelv típus nélküli általános mutatója. A **void*** típussal akkor definiálunk mutatókat, ha előtte nem tudjuk a megcímezendő adat típusát. Bővebben lásd a ***malloc()*** függvény használatánál.

- **Többszörös indirektségű mutatók**

`int x = 7, y;`	
`int *p;`	*p* egy int változóra mutató pointer.
`int **pp;`	**int*** változóra mutató pointer.
`p = &x; pp = &p;`	a címek átvétele.
`y = *p + **pp;`	*y* értéke 14 lesz.



A példában a **p* és a ***pp* végül a címzések által az *x* tartalmát érik el, mindkettő értéke 7 lesz, ezért az *y* a 7+7 összegét, azaz 14-et kapja értékül.

- **Típuskonverziók:**
A típuskonverziók egy része automatikusan – a programozó beavatkozása nélkül – megy végbe a C nyelv definíciójában rögzített szabályok alapján. Ezeket a konverziókat ***implicit*** vagy ***automatikus*** konverzióknak nevezzük.
Explicit típuskonverzióról beszélünk, ha a programozó típuskonverziót ír elő a C programban a típuskonverziós operátor (*cast*) felhasználásával.

(típusnév)kifejezés

```
/*MINTA4_01*/
#include <stdio.h>
int main()
{
  int i = 5, j, k = 2;
  double x = 4.5, ered1 = 0, ered2, ered3;
  ered1 = i + x; printf("eredmeny1: %lf\n",ered1);
  j = i + x; /* automatikus konverzió */
  printf("eredmeny2: %d\n",j);
  ered2 = i / k;
  printf("Egeszek osztasa tipuskonverzio nelkul: %lf\n",ered2);
  ered3 = (double)i / k; /* típuskonverzió */
  printf("Egeszek osztasa tipuskonverzioval: %lf\n",ered3);
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
eredmeny1: 9.500000
eredmeny2: 9
Egeszek osztasa tipuskonverzio nelkul: 2.000000
Egeszek osztasa tipuskonverzioval: 2.500000
```

- **Tömbváltozó deklarálása**
  A tömb azonos típusú értékeket tárol:

`double j[15];`	az *j* tömb egész elemeket tárol, melyek indexe 0-14-ig változhat.
`j[1] = 13;`	a *j* tömb 2. eleme kap értéket (mivel a *j[0]* eleme az elso).
`double y[100];`	az *y* tömb valós elemeket tárol, melyek indexe 0-99-ig változhat.
`y[0] = 2.4;`	az *y* tömb első eleme 2.4 értéket kap.

- **Dinamikus helyfoglalás**
  A C nyelvben szoros kapcsolat van a mutatók és az egydimenziós tömbök között. A mutatók segítségével a tömbök dinamikusan létrehozhatók, számukra a kívánt nagyságú területet a ***malloc()*** vagy a ***calloc()*** függvények segítségével foglalhatunk le:

  void * ***malloc***(**size_t** *méret*);
  void * ***calloc***(**size_t** *méret*, **size_t** *elemméret*);

  A ***calloc()*** függvény a lefoglalt területet 0-s bájtokkal tölti fel, ami főleg a sztringek esetén hasznos. A függvények használatához az *STDLIB.H* fejállományt kell a programba beszerkeszteni a *#inlcude* előfordító direktívával.

  A függvények a lefoglalt területre mutató általános pointert (**void***) adnak vissza, melyet a bal oldalnak megfelelő típusra kell átalakítani. Ha a visszatérési érték *NULL*, akkor a memóriaterület helyfoglalása nem sikerült.

```
double *ap=NULL;
```
ap **double** típusú pointer, amely **double** típusú tárolóra mutathat, de még nem tartalmaz címet, ezt a *NULL* értékkel jelöljük.

```
ap = (double*)malloc(sizeof(double));
```

```
if(ap != NULL)
```
Az *ap* már **double** adatra mutató címet tartalmazza.

```
*ap = 1.56;
```
**ap* értékadás a mutatóval kijelölt tárolónak.

```
int *xp=NULL;
```
Az *xp* **int** típusú pointer, amely **int** típusú tárolóra mutathat, de még címet nem tartalmaz, ezt *NULL* értékkel jelöljük.

```
xp =(int*)malloc(30 * sizeof(int));
```

```
if(xp != NULL)
```
A 30 darab **int** típusú adat számára foglaltunk helyet a memóriában, így egy 30-elemű tömböt hoztunk létre. A tömb indexelése 0-29 között lehetséges, mivel a C nyelvben minden tömb 0 indexű elemmel kezdődik.

```
xp[0]= 3;
```
Értékadás a tömb első, 0. indexű elemének.

```
xp[29]= 12;
```
Értékadás a tömb utolsó, 29. indexű elemének.

- **A dinamikusan lefoglalt memóriaterület felszabadítása**

  A ***free()*** függvény a ***malloc()*** vagy a ***calloc()*** függvény által lefoglalt memóriaterületet szabadítja fel.

```
free(ap);      /* az ap pointer által mutatott memóriaterület
               :   felszabadítása */
ap = NULL;
```

```
free(xp);      /*az xp tömb memóriaterületének felszabadítása */
xp = NULL;
```

A NULL értékkel jelezzük, hogy a pointerek most már sehova sem mutatnak, mivel a *free()* függvény nem nullázza azokat.

- **Mutatók használata**

Olvassuk be a henger sugarát és magasságát, számítsuk ki a henger felszínét és térfogatát! Használjuk mutatókat a feladat megoldásához!

```
/*MINTA4_02*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define PI 3.141592654
int main()
{
  double *r = NULL, *h = NULL, hengFelsz, hengTerf;
  r = (double*)malloc(sizeof(double));
  h = (double*)malloc(sizeof(double));
  if ( r == NULL || h == NULL)
  {
    printf("Memoriafoglalas nem sikerult!\n");
    getchar(); return 1;
  }
  printf("Henger sugara: ");
  scanf("%lf",r);
  printf("Henger magassaga: ");
  scanf("%lf",h);
  hengFelsz = 2 * pow(*r,2) * PI + 2 * *r *PI * *h;
  hengTerf = pow(*r,2) * PI * *h;
  printf("\nHenger felszine: %lf\n",hengFelsz);
  printf("Henger terfogata: %lf\n",hengTerf);
  free(r); free(h); r = NULL; h = NULL;
  fflush(stdin); getchar();
  return 0;
}
```

A program futásának eredménye:

```
Henger sugara: 1
Henger magassaga: 1

Henger felszine: 12.566371
Henger terfogata: 3.141593
```

Dinamikusan foglaljuk le a memóriahelyet egy valós elemeket tartalmazó tömb számára! Töltsük fel a tömböt adatokkal, és keressük meg a legkisebb és a legnagyobb elemét, írassuk ki az eredményt és szabadítsuk fel a tömb által elfoglalt memóriát!

```
/*MINTA4_03*/
#include <stdio.h>
#include <stdlib.h>
```

```
int main()
{ int i, db;
  double *y = NULL, Min, Max;
  printf("Elemek szama: "); scanf("%d",&db);
  y = (double*)malloc(db * sizeof(double));
  if (y == NULL)
  {
    printf("Memoriafoglalas nem sikerult!\n");
    fflush(stdin); getchar(); return 1;
  }
  for (i = 0; i<db; i++)
  {
    printf("%d. adat: ",i);
    scanf("%lf",&y[i]);
  }
  Min = y[0]; Max = y[0];
  for (i = 1; i<db; i++)
  {
    if (y[i] < Min) Min = y[i];
    if (y[i] > Max) Max = y[i];
  }
  printf("\nLegkisebb adat : %8.2lf\n",Min);
  printf("Legnagyobb adat: %8.2lf\n",Max);
  free(y);
  fflush(stdin); getchar();
  return 0;
}
```

A program futásának eredménye:

```
Elemek szama: 5
0. adat: 2.4
1. adat: 3.2
2. adat: 1.7
3. adat: 0.4
4. adat: 2.8

Legkisebb adat :    0.40
Legnagyobb adat:    3.20
```

Feladatok

1. Használjuk az alábbi definíciókat!

```
#define M 20

int x[M], n, paros = 0, paratlan = 0, osszeg = 0, i;
double atlag;
```

Olvassuk be az adatok számát az *n* változóba, majd töltsük fel az *x* tömböt egész adatokkal! Számítsuk ki a tömb elemeinek összegét, átlagát! Számláljuk meg a páros és a páratlan adatok darabszámát! (GYAK4_1)

A program futásának eredménye:

```
Elemek szama (max.20): 5
 0. adat: 1
 1. adat: 2
 2. adat: 3
 3. adat: 4
 4. adat: 5

Osszeg: 15
Atlag :     3.00
Paros elemek szama   : 2
Paratlan elemek szama: 3
```

2. Használjuk az alábbi definíciókat!

```
#define N 4
double a[] = {1.2, 2.4, 3.2, 4.7};
double b[] = {2.3, 5.3, 1.6, 6.4};
double kul[N], ossz[N];
```

Számítsuk ki az *a* és a *b* tömbök összegét az *ossz*, a különbségét mely a *kul* tömbbe, és írassuk ki az eredményt! (GYAK4_2)

A program futásának eredménye:

```
Kulonbseg
   -1.10      -2.90       1.60      -1.70

Osszeg
    3.50       7.70       4.80      11.10
```

3. Használjuk az alábbi definíciókat!

```
int db, i;
int *y, max, min;
double MinMaxAtlag;
```

Olvassuk be az adatok számát a *db* változóba! Foglaljuk le a memóriát az *y* tömb számára! Keressük meg tömb legkisebb és a legnagyobb elemét és számítsuk ki azok átlagát! (GYAK4_3)

A program futásának eredménye:

```
Elemek szama: 5
 0. elem: 1
 1. elem: 2
 2. elem: 3
 3. elem: 4
 4. elem: 5

Legkisebb adat : 1
Legnagyobb adat: 5
MinMaxAtlag:     3.00
```

4. Használjuk az alábbi definíciókat!

```
int db, i, j;
double *adatok, x;
```

Olvassuk be a rendezni kívánt adatok számát (*db*), foglaljuk le a memóriát az *adatok* tömb számára, majd olvassuk be a valós adatokat a tömbbe! Rendezzük a tömböt növekvő sorrendbe! (GYAK4_4)

A program futásának eredménye:

```
Elemek szama: 5
0. adat: 2.5
1. adat: 1.7
2. adat: 3.5
3. adat: 2.1
4. adat: 4.5
```

```
Novekvo sorrendben rendezett adatok
    1.70
    2.10
    2.50
    3.50
    4.50
```

5. Használjuk az alábbi definíciókat!

```
char aruAzonosito[10][20];
int darab[10];
double ara[10], osszAr = 0;
int db, i, osszDb = 0;
```

Számítsuk ki a raktárban lévő áruk összértékét és az áruk összes darabszámát! (GYAK4_5)

A program futásának eredménye:

```
Arufajtak szama: 3

Aru azonositoja: Asztal
Aru darabszama: 3
Aru ara: 1500

Aru azonositoja: Szek
Aru darabszama: 4
Aru ara: 500

Aru azonositoja: Szekreny
Aru darabszama: 2
Aru ara: 5000

Osszes darabszama: 9
Osszerteke: 16500.00
```

5. Gyakorlat

Rövid elméleti összefoglaló

Felhasználó által definiált adattípusok:

- A **typedef** egy speciális tárolási osztály, mellyel érvényes típusokhoz szinonim nevet rendelhetünk.

  typedef *létező_típus felhasználói_típus*;

```
typedef  double  tomb[10];
tomb  x;
```

  A *tomb* olyan tömbtípus, amelynek 10 eleme (0-9 között indexelve) van, és minden eleme valós.

- **Struktúra – összetartozó információk tárolása**
  A struktúra több tetszőleges típusú (kivéve a **void** és függvénytípust) adatok együttese. Általános formája:

```
struct struktúra_azonosító {  /* típusnév */
                              tipus1 tag;
                                ...
                           };
```

  Megjegyezzük, hogy a struktúra deklarációját záró kapcsos zárójel mögé kötelező a pontosvesszőt kitenni. A fenti típussal struktúraváltozót készíthetünk:

  struct *struktúra_azonosító struktúra_változó*;

  A *struktúra_azonosító* a **struct** kulcsszóval együtt jelöli az új felhasználói típust.

```
/* raktar struktúra definiálása */
struct raktar {
                  char aru[20];
                  int darab;
                  double ara;
                };
```

  Definiáljunk néhány változót a **struct** *raktar* típus felhasználásával:

```
struct raktar r1, *r2;
```

  A statikus helyfoglalású *r1* változó adattagjait a (.) operátorral érjük el:

```
strcpy(r1.aru,"asztal"); r1.darab = 2; r1.ara = 1200;
```

  Dinamikus memóriaterület foglalása az *r2* mutató számára a ***malloc()*** függvénnyel:

```
r2 = (struct raktar*)malloc(sizeof(struct raktar));
```

  Kétféle módon adhatunk értéket az *r2* által kijelölt struktúra adattagjainak:

  - a nyíl operátor (->) biztosítja az adattag elérését:

```
strcpy(r2->aru,"szekreny");r2->darab = 10;
```

  - Amikor a pont operátort az *r2* által mutatott objektumra alkalmazzuk, a precedencia-szabályok miatt zárójelek között kell megadnunk a **r2* kifejezést:

```
(*r2).ara = 15000;
```

A mutatóval kijelölt memóriaterületet fel kell szabadítanunk a *free()* függvénnyel!

```
free(r2);
```

Karaktertömbnek az *strcpy()* függvénnyel adhatunk értéket, melynek használatához a *STRING.H* fejlécállomány szükséges.

```
/*MINTA5_01*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct raktar{
                char aru[20];
                int darab;
                double ara;
              };
int main()
{ struct raktar r1, *r2;
  strcpy(r1.aru,"asztal");
  r1.darab = 2;
  r1.ara = 1200;
  r2 = (struct raktar*)malloc(sizeof(struct raktar));
  if (r2 == NULL)
  { printf("Memoriafoglalas nem sikerult!\n");
    getchar(); return 1;
  }
  strcpy(r2->aru, "szekreny");
  r2->darab = 10;
  (*r2).ara = 15000;
  printf("aru: %-10s darab: %4d ara: %9.2lf\n",
         r1.aru, r1.darab, r1.ara);
  printf("aru: %-10s darab: %4d ara: %9.2lf\n",
         r2->aru, r2->darab, r2->ara);
  free(r2); getchar();
  return 0;
}

A program futásának eredménye:
aru: asztal     darab:    2 ara:   1200.00
aru: szekreny   darab:   10 ara:  15000.00
```

Megjegyzés: A kiírásban a "%-10s" formátum az áru azonosítóját 10 mezőszélességben, balra igazítva jeleníti meg. A "%4d" formátum az áru darabszámát 4 mezőszélességgel jobbra igazítva, a "%9.2lf" pedig az áru árát 9 mezőszélességben 2 tizedes jeggyel jobbra igazítva jeleníti meg.

```
/* struktúra típus definiálása */
typedef struct raktar {
                        char aru[20];
                        int darab;
                        double ara;
                       }Raktar;
Raktar rak1;
struct raktar *rak3;
```

Struktúra feltöltése konstans kezdőértékekkel.

A ***struct** raktar* és a *Raktar* ugyanazt a struktúratípust jelöli (*MINTA5_02*).

```
Raktar rak1 = {"szek",3,600};  /* konstans kezdőértékek */
```

```
/*MINTA5_02*/
#include <stdio.h>
#include <stdlib.h>
typedef struct raktar{ char aru[20];
                       int darab;
                       double ara;
                      }Raktar;
Raktar rak1 = {"szek",3,600}; /* konstans kezdőértékek */
int main()
{ Raktar *rak2;
  rak2 = (Raktar*)malloc(sizeof(Raktar));
  if (rak2 == NULL)
  { printf("Memoriafoglalas nem sikerult!\n");
    getchar(); return 1;
  }
  strcpy(rak2->aru, "szekreny");
  rak2->darab = 10;
  (*rak2).ara = 15000;
  printf("aru: %-10s darab: %4d ara: %9.2lf\n",
         rak1.aru, rak1.darab, rak1.ara);
  printf("aru: %-10s darab: %4d ara: %9.2lf\n",
         rak2->aru, rak2->darab, rak2->ara);
  free(rak2); getchar();
  return 0;
}
```

A program futásának eredménye:

```
aru: szek       darab:    3 ara:     600.00
aru: szekreny   darab:   10 ara:   15000.00
```

- **Struktúratömb használata.**

  Példaként használjuk fel a *Raktar* struktúrát és az alábbi definíciót:

  ```
  Raktar *rak1;
  int db; /* árufféleségek száma */
  ```

  Rákérdezünk az árufajtákra, és beolvassuk a darabszámot a *db* változóba:

  ```
  printf("Arufajtak szama: "); scanf("%d",&db);
  ```

  A struktúratagok számára helyet foglalunk a dinamikusan kezelt memóriaterületen, a *db* változó tartalmazza az elemek számát:

  ```
  rak1 = (Raktar*)malloc(db * sizeof(Raktar));
  ```

  A **sizeof**(*Raktar*) kifejezés megadja a *Raktar* struktúra memóriaterület foglalását bájtban, és mivel *db* darab eleme lesz a tömbnek, így a szükséges helyet db * **sizeof**(Raktar) szorzat adja. A ***malloc()*** függvény foglalja le a kívánt memóriaterületet. A visszaadott **void*** típusú pointert a *Raktar** mutatóvá átalakítva kapja meg a *rak1* változó, amely érték a struktúratömb 0. elemére mutató címet jelenti.

```
/*MINTA5_03*/
#include <stdio.h>
#include <stdlib.h>
typedef struct raktar{
                        char aru[20];
                        int darab;
                        double ara;
                       }Raktar;
int main()
{
  Raktar *rak1;
  int i, db, mindb, minIndex, maxIndex;
  double maxar;
  printf("Arufajtak szama: "); scanf("%d",&db);
  rak1 = (Raktar*)malloc(db * sizeof(Raktar));
  if (rak1 == NULL)
  {
    printf("Memoriafoglalas nem sikerult!\n");
    fflush(stdin); getchar(); return 1;
  }
  for (i = 0; i<db; i++)
  {
    printf("\nAru azonositoja: "); scanf("%s",rak1[i].aru);
    printf("Aru darabszama: "); scanf("%d",&rak1[i].darab);
    printf("Aru ara: "); scanf("%lf",&rak1[i].ara);
  }
  mindb = rak1[0].darab;
  maxar = rak1[0].ara;
  minIndex = 0; maxIndex = 0;
  for (i = 1; i<db; i++)
  {
    if (rak1[i].darab < mindb)
    {
      mindb = rak1[i].darab;
      minIndex = i;
    }
    if (rak1[i].ara > maxar)
    {
      maxar = rak1[i].ara;
      maxIndex = i;
    }
  }
  printf("\nLegkisebb darabszamu aru\n");
  printf("aru: %-10s darab: %4d ara: %9.2lf\n",
  rak1[minIndex].aru, rak1[minIndex].darab, rak1[minIndex].ara);
  printf("Legdragabb aru\n");
  printf("aru: %-10s darab: %4d ara: %9.2lf\n",
  rak1[maxIndex].aru, rak1[maxIndex].darab, rak1[maxIndex].ara);
  free(rak1);
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
Arufajtak szama: 3

Aru azonositoja: szek
Aru darabszama: 5
Aru ara: 500

Aru azonositoja: asztal
Aru darabszama: 2
Aru ara: 1000

Aru azonositoja: szekreny
Aru darabszama: 6
Aru ara: 2500

Legkisebb darabszamu aru
aru: asztal     darab:     2 ara:    1000.00
Legdragabb aru
aru: szekreny   darab:     6 ara:    2500.00
```

Feladatok

1. Használjuk fel az alábbi definíciót és deklarációt!

```
struct Teglatest{
                    double a, b, c;
                    double terf, felsz;
                  };
int main()
{
  struct Teglatest t;
  . . .
}
```

Olvassuk be a **struct** *Teglatest* típusú *t* struktúra három valós adatát (a téglatest oldalainak hosszát), és számítsuk ki a téglatest felszínét és a térfogatát! (GYAK5_1)

A program futásának eredménye:

```
A oldal: 2
B oldal: 3
C oldal: 4
Teglatest felszine : 52.000000
Teglatest terfogata: 24.000000
```

2. Módosítsuk az 1. feladatot, adjuk meg kezdőértékként a téglatest három oldalának hosszát! (GYAK5_2)

```
struct Teglatest{
                    double a, b, c;
                    double terf,felsz;
                  };
struct Teglatest t = { 1, 2, 3};
```

A program futásának eredménye:

```
Teglatest oldalai   :  1.00  2.00  3.00
Teglatest felszine :  22.00
Teglatest terfogata:   6.00
```

3. Használjuk az alábbi definíciót és deklarációt!

```
#include <stdio.h>
#include <math.h>
#define PI 3.141592654
typedef struct kr { double sugar;
                    double ter, ker;
                  } Kor;
int main()
{ Kor k;
  . . .
}
```

Olvassuk be a *Kor* típusú *k* struktúra sugar adatát! Számítsuk ki a kör területét és kerületét. A PI értékét adjuk meg *#define* előfordító utasítással! (GYAK5_3)

A program futásának eredménye:

```
sugar: 1
Kor kerulete: 6.283185
Kor terulete: 3.141593
```

4. Használjuk az alábbi definíciót és deklarációt!

```
typedef struct k {  double adat1, adat2;
                    double mertanikozep;
                  } Adatok;
int main()
{ Adatok *pa;
  . . .
}
```

Foglaljunk memóriaterületet a *pa* által mutatott *Adatok* struktúra számára. Olvassunk be a két adattagot, és számítsuk ki a két adat mértani közepét! (GYAK5_4)

5. Használjuk az alábbi definíciót és deklarációt!

```
typedef struct k {  double adat1, adat2;
                    double atlag;
                  } Szamtanikozep;
int main()
{ Szamtanikozep *szk;
  double OsszAtlag = 0;
  int i, db;
  . . .
}
```

Foglaljunk memóriaterületet a *db* darab *Szamtanikozep* struktúra számára, olvassuk be a két adattagot, és számítsuk ki a két adat átlagát, valamint az átlagok összegét! (GYAK5_5)

6. Gyakorlat

Rövid elméleti összefoglaló

- **Függvények deklarációja és definíciója**

A saját függvényeinket mindig definiálni kell. A definíció, amelyet csak egyszer lehet megadni, a C programon belül bárhol elhelyezkedhet. Ha a függvény definíciója megelőzi a függvény hívásának helyét, akkor ez helyettesíti a függvény prototípusának megadását. A prototípus: `<visszatérési típus> függvénynév(paraméterek típusa);`

A függvény definíció általános formája:

<visszatérési típus> függvénynév(paraméterdeklarációs lista)
{ / függvény törzse */*
<lokális definíciók és deklarációk >
< utasítások>
}

Ha a visszatérési típus **void**, akkor a függvénynek nem kell tartalmaznia **return** utasítást, mivel nem ad vissza semmit.

Ellenkező esetben a **return** melletti kifejezés típusának meg kell egyeznie (különben konverziót kell alkalmaznunk) a függvény visszatérési típusával. Ebben az esetben a függvénynek ez a visszatérési értéke, vagyis a függvény működésének eredménye.

A függvény paraméterlistáján a paraméterek megadása: a paraméter típusa a paraméter nevével (vesszővel tagolva, ha több paraméter is van), például: `int n, double c` stb. A paraméterlistán adjuk meg a bemenő, és a kimenő paramétereket. Átíráskor a bemenő paramétereknek kell, hogy értékük legyen, míg a kimenő paraméterek a függvény törzsében kapnak értéket. Csak mutatóval átadott paraméter segítségével adhatunk ki értéket függvényből.

```
double atlag1(double x, double y)
{
  double m;
  m = (x + y)/2;
  return m;
}
```

Az *atlag1()* függvénynek két valós típusú bemenő paramétere van: az *x* és az *y*. A függvény feladata, hogy a két bemenő paraméter átlagát adja vissza visszatérési értékként.

Lokálisan definiálunk egy valós típusú *m* változót. Az *m* változóba adjuk össze az *x* és a *y* értékeket, és elosztjuk el kettővel, majd a **return** utasítással visszaadjuk az *m* tartalmát.

```
double atlag2(double x, double y)
{
  return (x + y)/2;
}
```

A feladatot így is megoldhatjuk:
Az egyszerűbb számítást tartalmazó kifejezést a **return** mellett közvetlenül is megadhatjuk.

```
void atlag3(double x, double y,
            double *szk)
{
  *szk = (x + y)/2;
}
```

Az *atlag3()* függvényben az eredményt a paraméterlistán **double** típusra mutató *szk* pointer segítségével adjuk vissza.

Készítsünk a fenti függvények használatához ***main()*** függvényt! (*MINTA6_01*)

Javasolt programozási módszer:

- A függvények prototípusát a ***main***() függvény előtt adjuk meg.
- A függvényeket a ***main***() függvény után dogozzuk ki.

```
/*MINTA6_01*/
#include <stdio.h>  /* printf használatára */
#include <stdlib.h> /* malloc() függvény használatára */
/*A függvények prototípusa */
double atlag1(double x, double y);
double atlag2(double x, double y);
void atlag3(double x, double y, double *szk);
int main()
{ double a = 14, b = 8, szk1, szk2, szk3, *szk4;
  szk1 = atlag1(a, b);
  szk2 = atlag2(a, b);
  atlag3(a, b, &szk3); /* az szk3 címét kell átadni */
  /* az szk4 pointer számára memóriaterület foglalása */
  szk4 = (double*)malloc(sizeof(double));
  if (szk4 == NULL)
  { printf("Memoriafoglalas nem sikerult!\n");
    getchar(); return 1;
  }
  atlag3(a, b, szk4);
  printf("1. adat: %8.2lf\n2. adat: %8.2lf\n",a,b);
  printf("\n1. szamtani kozep: %8.2lf\n",szk1);
  printf("2. szamtani kozep: %8.2lf\n",szk2);
  printf("3. szamtani kozep: %8.2lf\n",szk3);
  printf("4. szamtani kozep: %8.2lf\n",*szk4);
  free(szk4); getchar();
  return 0;
}
/* Számtani közép számítása függvényekkel */
double atlag1(double x, double y)
{
  double m;
  m = (x + y)/2;
  return m;
}
double atlag2(double x, double y)
{  return (x + y)/2; }
void atlag3(double x, double y, double *szk)
{  *szk = (x + y)/2; }
```

A program futásának eredménye:

```
1. adat:    14.00
2. adat:     8.00

1. szamtani kozep:    11.00
2. szamtani kozep:    11.00
3. szamtani kozep:    11.00
4. szamtani kozep:    11.00
```

Az adatoknak függvénnyel adjunk értéket és használjunk pointeres változót bemenő paraméterként! (*MINTA6_02*)

```
/*MINTA6_02*/
#include <stdio.h>
#include <stdlib.h>
void olvas(double *x, double *y);
void atlag(double x, double y, double *szk);
int main()
{
  double a1, b1, szk1, *a2 = NULL, *b2 = NULL, *szk2 = NULL;
  olvas(&a1, &b1); /* adatok beolvasása */
  atlag(a1, b1, &szk1); /* átlag számítása statikus változókkal */
  printf("Szamtani kozep: %8.2lf\n\n",szk1);
  a2 = (double*)malloc(sizeof(double));
  b2 = (double*)malloc(sizeof(double));
  szk2 = (double*)malloc(sizeof(double));
  if (a2 == NULL || b2 == NULL || szk2 == NULL)
  {
    printf("Memoriafoglalas nem sikerult!\n");
    getchar(); return 1;
  }
  olvas(a2, b2);
  atlag(*a2, *b2, szk2);
  printf("Szamtani kozep: %8.2lf\n",*szk2);
  free(a2);free(b2);free(szk2);
  fflush(stdin);getchar();
  return 0;
}
```

```
void olvas(double *x, double *y)
{
  printf("1. adat: "); scanf("%lf",x);
  printf("2. adat: "); scanf("%lf",y);
}
```

Az *olvas()* függvény két valós adatnak ad értéket, melyet a paraméterlistán mutatók segítségével ad vissza. A ***scanf()*** függvény beírásakor az *x* és *y* változók címet tartalmaznak, ezért nincs szükség az & címe operátorra.

```
void atlag(double x, double y,
           double *szk)
{
  *szk = (x + y)/2;
}
```

Az *atlag()* függvény a két bemenő (*x,y*) valós paraméternek az átlagát adja vissza az *szk* **double**-ra mutató kimenő paraméterrel.

A program futásának eredménye:

```
1. adat: 1
2. adat: 2
Szamtani kozep:     1.50

1. adat: 3
2. adat: 4
Szamtani kozep:     3.50
```

- **Tömbök átadása függvénynek**

A tömböt nem lehet érték szerint átadni a függvénynek, illetve függvényértékként megkapni. Az **double**[ ] típusú vektorargumentum **double*** típusú mutatóként adódik át a függvénynek. A függvényen belül a tömbben végrehajtott változások a függvényen kívül is megmaradnak.

```
/* 10 elemű valós tömb adott számú elemének adjunk értéket és
számítsuk ki az tömb elemeinek összegét. */
/*MINTA6_03*/
#include <stdio.h>
void olvas_tomb(double a[], int *db);
double osszegez(double a[], int db);
int main()
{ int n;
  double x[10];
  double osszeg;
  olvas_tomb(x,&n);
  osszeg = osszegez(x,n);
  printf("Az adatok osszege: %8.2f\n",osszeg);
  fflush(stdin); getchar();
  return 0;
}
/* Az olvas() függvény feladata a paraméterként kapott a tömb a
függvényben megadott darabszámú adattal való feltöltése. */
void olvas_tomb(double a[], int *db)
{
   int i;
   printf("A tomb elemeinek szama: ");
   scanf("%d",db);
   for (i = 0; i<*db; i++)
   {
     printf("%d. adat : ",i);
     scanf("%lf",&a[i]);
   }
}
/* Az osszegez() függvény a tömb (a) és a tömb elemeinek száma (db)
ismeretében kiszámítja a tömb elemeinek összegét az ossz változóban,
melyet függvényértékként a return mellett adja vissza. */
double osszegez(double a[], int db)
{ int i;
  double ossz = 0;
  for( i = 0; i<db; i++)
    ossz += a[i];
  return ossz;
}
```

A program futásának eredménye:

```
tomb elemeinek szama: 3
. adat : 4
. adat : 2
. adat : 6
Az adatok osszege:    12.00
```

- **Típusdefinícióval megadott tömb a függvényparaméter**

A **typedef** utasítással tömbtípust definiálhatunk:

```
typedef double tomb[10];
```

A *tomb* olyan tömb típus, melynek [0-9] indextartományban vannak valós típusú elemei.

```
/* A mintafeladat bemutatja a tömbtípus használatát. A tömbtípussal
definiált tömb visszatérési értékként nem adható meg! */
/*MINTA6_04*/
#include <stdio.h>
typedef double tomb[10];
void OlvasTomb(tomb a, int *db);
double AtlagSzamol(tomb a, int db);
int main()
{ int n;
  tomb x;
  double atlag;
  OlvasTomb(x,&n);
  atlag = AtlagSzamol(x,n);
  printf("Az adatok atlaga: %8.2f\n",atlag);
  fflush(stdin); getchar();
  return 0;
}
/* OlvasTomb() a paraméterlistán adja vissza a tomb elemeinek
darabszámát (*db) és az adatokkal feltöltött a tömböt */
void OlvasTomb(tomb a, int *db)
{  int i;
   printf("A tomb elemeinek szama: ");
   scanf("%d",db);   /* beolvassuk az elemek számát a db pointerbe */
   for (i = 0; i<*db; i++)
   { printf("%d. adat : ",i);
     scanf("%lf",&a[i]);  /* a tombot feltöltjük adatokkal */
   }
}
/* AtlagSzamol() az a tömb és a db darabszám ismeretében kiszámítja a
tömb elemeinek átlagát*/
double AtlagSzamol(tomb a, int db)
{ int i;
  double atlag;
  for( i = 0, atlag = 0; i<db; i++) /* atlag 0 kezdőértéket kap */
  {  atlag += a[i]; } /* hozzáadjuk a tömb elemeit */
  atlag /= db;        /* elosztjuk a tömb elemeinek számával */
  return atlag;
}
```

A program futásának eredménye:

```
A tomb elemeinek szama: 4
0. adat : 2
1. adat : 5
2. adat : 3
3. adat : 6
Az adatok atlaga:     4.00
```

- **A mutató típusú függvényparaméter**

Mutatóval megadott tömbnek dinamikusan foglaljunk memóriahelyet, töltsük fel adatokkal, és a tömb legnagyobb elemével osszuk el a tömb elemeit! A feladatot függvényekkel oldjuk meg! A pointerként megadott tömböt a függvény visszatérési értékként adja vissza:

```
double* OlvasTomb(int *db);
```

```
/*MINTA6_05*/
#include <stdio.h>
#include <stdlib.h>
double* OlvasTomb(int *db);
double NormalMax(double *a, int db);
int main()
{
 int n;
 double *x;
 double MaxErtek;
 x = OlvasTomb(&n); /*az n változó címével helyettesítjük a pointert */
 if (x == NULL)
 { printf("Memoriafoglalas nem sikerult!\n"); getchar(); return 1; }
 MaxErtek = NormalMax(x,n);
 printf("Legnagyobb ertek: %8.2f\n",MaxErtek);
 free(x);
 fflush(stdin);getchar();
 return 0;
}

double* OlvasTomb(int *db) /* a darabszám a kimenő paraméter */
{
   int i;
   double *y;
   printf("A tomb elemeinek szama: ");
   scanf("%d",db); /* a tömbelem darabszámának beolvasása */
   y = (double*)malloc(*db * sizeof(double));
   if (y != NULL)
   {
     for (i = 0; i<*db; i++)
     {
       printf("%d. adat : ",i);
       scanf("%lf",&y[i]); /* tömb elemeinek feltöltése adatokkal */
     }
    }
   return y; /* a tömb első elemére mutató cím a visszatérési érték */
}
double NormalMax(double *a, int db)
{
  int i;
  double Max = a[0];
  for( i = 1; i<db; i++)
  {
    if (a[i] > Max) Max = a[i]; /* legnagyobb adat kikeresése */
  }
  printf("\nNormalt adatok\n");
```

```
  for (i = 0; i<db; i++)
  {
    a[i] /= Max;     /* normálás a legnagyobb adattal */
    printf("%9.4lf\n",a[i]); /* normált adatok megjelenítése */
  }
  return Max;
}
```

A program futásának eredménye:

```
A tomb elemeinek szama: 4
0. adat : 4
1. adat : 2
2. adat : 8
3. adat : 6

Normalt adatok
   0.5000
   0.2500
   1.0000
   0.7500
Legnagyobb ertek:      8.00
```

Megjegyzés: a paraméterlistán mutatót is vissza lehet adni (pl. **double** **a). Példaként nézzük meg a *MINTA6_05* feladatban bemutatott *OlvasTomb()* függvény módosított változatát!

```
void OlvasTomb(double **a, int *db);
```

```
void OlvasTomb(double **a, int *db);
int main()
{
 int n;
 double *x;

 OlvasTomb(&x, &n);
 . . .
}
void OlvasTomb(double **a, int *db)
{
   int i;
   double *y = NULL;
   printf("A tomb elemeinek szama: ");
   scanf("%d",db);
   y = (double*)malloc(*db * sizeof(double));
   if (y != NULL)
   {
     for (i = 0; i<*db; i++)
     {
       printf("%d. adat : ",i);
       scanf("%lf",&y[i]);
     }
   }
   *a = y;
}
```

Az *OlvasTomb()* az *a* paraméter **double** * típusú mutató címét csak **double** ** típusú paraméterben lehet visszaadni.

Az *OlvasTomb()* aktiválásakor a **x* pointernek a címét (*&x*) kell átadni, hogy a függvény az *y* lokális változóban lefoglalt memóriaterület címét vissza tudja adni (**a = y*).

- **Struktúra átadása függvénynek**

Ha a függvényben a struktúra bármelyik adattagján változtatni szeretnénk, akkor a struktúrát mutatójával kell átadnunk, például: `Mkozep *sz`. Ebben az esetben nem a cím módosul, hanem a struktúra adattagja változik.

```
/*MINTA6_06*/
#include <stdio.h>
#include <math.h>
/* Az Mkozep struktúra két valós adatot és a két adat mértani közepét
tároló ered adattagot tartalmazza. */
typedef struct {
                 double x, y; /* adatok */
                 double ered; /* eredmény */
                } MKozep;

/* A függvények prototípusai */
void Olvas(MKozep *sz);
void Szamol(MKozep *sz);
int main()
{
  MKozep mk;
  Olvas(&mk);  /* struktúra adatainak beolvasása */
  Szamol(&mk); /* mértani közép számítása */
  printf("Mertani kozep: %lf\n",mk.ered);
  fflush(stdin); getchar();
  return 0;
}
void Olvas(MKozep *sz)
{
  printf("1. adat: ");  /* két adattag beolvasása */
  scanf("%lf",&sz->x);
  printf("2. adat: ");
  scanf("%lf",&sz->y);
}
void Szamol(MKozep *sz)
{
  sz->ered = sqrt(sz->x * sz->y); /* mértani közepet számol */
}
```

Megjegyzések a ***main()*** *függvényhez:*

Az *Olvas()* függvényt aktiváljuk az *mk* változóval, amely egy *MKozep* típusú struktúra. Ez a függvény ad értéket az *x* és az *y* adattagoknak.

A *Szamol()* függvény az adatokkal feltöltött *MKozep* típusú *mk* struktúra változót módosítja a kiszámított eredménnyel, majd az eredményt kiíratjuk.

A program futásának eredménye:

```
1. adat: 1
2. adat: 2
Mertani kozep: 1.414214
```

- **Struktúratömb átadása függvénynek**

Ha egy pointer struktúra típusra mutat, akkor a pointerhez rendelhetünk *n* darab struktúra hosszúságú területet, így *n*-elemű struktúratömböt kapunk, azaz a tömb minden eleme struktúra.

A MINTA6_07 feladat bemutatja a struktúratömb létrehozását, melynek elemei *Negyzet* típusú struktúrák. Beolvasva a négyzet oldalait, kiszámítja a négyzet területét, majd megkeresi a legnagyobb területű négyzetet.

```
/*MINTA6_07*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
typedef struct {
                  double a, ter;
                } Negyzet;
Negyzet* NOlvas(int *db);
void Keres(Negyzet *t, int darab, double *MaxTer);
int main()
{ Negyzet *w;
  int darab;
  double MaxTerulet;
  w = NOlvas(&darab);
  if (w == NULL)
  { printf("Memoriafoglalas nem sikerult!\n");
    fflush(stdin); getchar(); return 1;
  }
  Keres(w, darab, &MaxTerulet);
  printf("\nLegnagyobb terulet: %lf\n",MaxTerulet);
  free(w); fflush(stdin);getchar();
  return 0;
}
Negyzet* NOlvas(int *db)
{ int i;
  Negyzet *t = NULL;
  printf("Negyzetek szama: "); scanf("%d",db);
  t = (Negyzet*)malloc(*db * sizeof(Negyzet));
  if (t != NULL)
  {
    for (i = 0; i<*db; i++)
    {
      printf("%d. oldala: ",i); scanf("%lf",&t[i].a);
      t[i].ter = t[i].a * t[i].a;
    }
  }
  return t;
}
void Keres(Negyzet *t, int darab, double *MaxTer)
{ int i;
  *MaxTer = t[0].ter;
  for (i = 1; i<darab; i++) /* a legnagyobb terület keresése */
  { if (t[i].ter > *MaxTer) *MaxTer = t[i].ter; }
}
```

A program futásának eredménye:

```
Negyzetek szama: 3
0. oldala: 1
1. oldala: 6
2. oldala: 3

Legnagyobb terulet: 36.000000
```

- **Karaktersorozat-argumentumok**

Karaktersorozatot függvénynek **char** * típusú mutató, illetve egydimenziós karaktertömb segítségével egyaránt átadhatunk. Eltérően a tömbök átadásától, általában nem szükséges megadnunk a karakterek számát, hisz a sztringet egy nullás bájt zárja. A *MINTA6_08* mintafeladatban a karaktersorozat betűit nagybetűssé alakító *Nagybetus()* függvényben mutatót használunk, míg a rendszám formátumát ellenőrző *RendszamE()* függvényben a tömbös megoldást alkalmazzuk.

```
/*MINTA6_08*/
#include <stdio.h>
#include <ctype.h>
char * Nagybetus(char * s);
int RendszamE(const char rsz[]);
int main()
{
  char rsz[132] = "NAT-123";
  printf("Kerek egy rendszamot: ");
  gets(rsz);
  if (RendszamE(Nagybetus(rsz)))
    printf("A %s rendszam.\n", rsz);
  else
    printf("A %s nem rendszam.\n", rsz);
  getchar();
  return 0;
}
/* Karaktersorozat nagybetűssé alakítása */
char * Nagybetus(char * s)
{
   char *p = s;        /* mutató a sztring bejárására */
   while (*p)          /* míg el nem érjük a sztring végét ... */
     *p++ = toupper(*p);
   /* a sztring elejére mutató pointerrel térünk vissza */
   return s;
}
/* Ellenőrzi, hogy a megadott karaktersorozat rendszám-e
   a karaktersorozat hossza 7, az elso harom karakter nagybetű,
   melyet egy minuszjel, majd 3 decimális számjegy követ.      */
int RendszamE(const char rsz[])
{
   int i;
   if (strlen(rsz) != 7)
      return 0;
```

```
   for (i=0; i<3; i++)                    /* ABC */
     if (!isupper(rsz[i]))
       return 0;

   if (rsz[3]!='-')                       /*  -  */
     return 0;

   for (i=4; i<7; i++)                    /* 000 */
     if (!isdigit(rsz[i]))
       return 0;
   return 1;
}
```

A program futásainak eredménye:

```
Kerek egy rendszamot: ELU-017
A ELU-017 rendszam.

Kerek egy rendszamot: DCA+036
A DCA+036 nem rendszam.
```

- **Függvényszerű makrók készítése**

A makrók felhasználási lehetőségeit lényegesen megnövelik a paraméterezett makrók, melyek definíciójának általános formája:

#define *azonosító* (*paraméterlista*) *helyettesítő szöveg*

A makró hívása:

azonosító(*argumentumlista*)

A makróhívásban szereplő argumentumok számának meg kell egyeznie a definícióban szereplő paraméterek számával. (Lehet paraméterek nélküli makrót is definiálni!)

Átlag és mértani közép számítása makróval. (*MINTA6_09*)

```
/*MINTA6_09*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define atlag(a,b)(((a) + (b))/2.)
#define mkozep(a,b)(sqrt(fabs((a)*(b))))
int main()
{
 double a = -1, b = 2, szk, mk;
 szk = atlag(a, b);
 mk = mkozep(a, b);
 printf("Szamtani kozep: %lf\n",szk);
 printf("Mertani kozep : %lf\n",mk);
 szk = atlag(a+2, 3*b);
 printf("Szamtani kozep: %lf\n",szk);
 getchar();
 return 0;
}
```

A program futásának eredménye:

```
Szamtani kozep: 0.500000
Mertani kozep : 1.414214
Szamtani kozep: 3.500000
```

- **Függvényre mutató pointer átadása függvénynek**

Az *fv* olyan függvény, amelynek paramétere **double** és visszatérési értéke **double**:

```
typedef double fv(double x);
```

A *pfv* az *fv*-re mutató pointer típusa:

```
typedef fv *pfv;
```

Az *fg1()* függvény bemenő paramétere a szög értéke fokban, visszatérési érték a radiánná átalakított szög szinusza.

```
double fg1 (double x);
```

Az *fg2()* függvény bemenő paramétere a szög értéke fokban, visszatérési érték a radiánná átalakított szög szinusz és koszinusz szorzatának kétszerese.

```
double fg2 (double x)
```

A *tablazat()* függvény bemenő paramétere *xk* - kezdőérték, az *xv* - végérték, a *dx* - a lépésköz, *sz[]* karaktertömb, a táblázat fejléce, *f* a függvényre mutató pointer.

```
void tablazat(double xk, double xv, double dx, char sz[], pfv f);
```

```
/*MINTA6_10*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define PI 3.141592654
typedef double fv(double x);
typedef fv *pfv;
double fg1 (double x);
double fg2 (double x);
void tablazat(double xk, double xv,
              double dx, char sz[], pfv f);
int main()
{
  char sz[40];
  strcpy(sz,"      x          sin(x) ");
  tablazat(0,80,10,sz,fg1);
  strcpy(sz,"      x   2*sin(x)*cos(x) ");
  tablazat(0,80,10,sz,fg2);
  getchar();
  return 0;
}

double fg1 (double x)
{ return sin(x*PI/180); }

double fg2 (double x)
{ return 2*sin(x*PI/180)*cos(x*PI/180); }

void tablazat(double xk, double xv,
              double dx, char sz[], pfv f)
{
```

A program futásának eredménye:

```
     x          sin(x)
  0.00          0.00
 10.00          0.17
 20.00          0.34
 30.00          0.50
 40.00          0.64
 50.00          0.77
 60.00          0.87
 70.00          0.94
 80.00          0.98

     x    2*sin(x)*cos(x)
  0.00          0.00
 10.00          0.34
 20.00          0.64
 30.00          0.87
 40.00          0.98
 50.00          0.98
 60.00          0.87
 70.00          0.64
 80.00          0.34
```

```
  double xx, yy;
  xx = xk;
  printf("%s\n",sz);
  while (xx <= xv)
  {
    yy = (*f)(xx); /*azonos az f(xx) hívással*/
    printf("  %8.2lf  %8.2lf\n",xx,yy);
    xx += dx;
  }
}
```

Feladatok

1. Tervezzünk olyan *Olvas()* függvényt, amely beolvas három valós adatot, és *KeresMinMax()* függvényt, amely a paraméterlistán megkapott három valós argumentumból kiválasztja a legnagyobb értékű adatot, melyet a paraméterlistán, a legkisebb értékű adatot függvényértékként adja vissza. Írjunk hozzá ***main()*** függvényt! (GYAK6_1)

```
void Olvas(double *x1, double *x2, double *x3);
double KeresMinMax(double x1, double x2, double x3, double *Max);
int main()
{
  double a, b, c, Legkisebb, Legnagyobb;
  ...
}
```

A program futásának eredménye:

```
1. adat: 3
2. adat: 1
3. adat: 5

Legkisebb  :     1.00
Legnagyobb :     5.00
```

2. Használjuk az alábbi definíciókat és deklarációkat!

```
typedef int vekt[50];
int Olvas(vekt x);
int OsszegSzamol(vekt x, int n);
void Vizsgal1(vekt x, int n, int *poz, int *neg, int *zero);
int Vizsgal2(vekt x, int n, int *paratl);
void AtlagSzamol(vekt x, int n, double *atl);
int main()
{
  vekt w;
  int db, ossz, pozitiv, negativ, zerus, paros, paratlan;
  double atlag;
  . . .
}
```

Írjunk *Olvas()* függvényt, amely a paraméterlistán adja vissza az *x* tömb adatait és függvényértékként a tömb aktuális darabszámát! Az *OsszegSzamol()* függvény függvényérték-

ként adja vissza a tömb elemeinek összegét. A *Vizsgal1()* függvényt a tömb elemeinek és darabszámának ismeretében megszámlálja a tömb pozitív, negatív és zérus elemeinek számát. A *Vizsgal2()* függvény a tömb páratlan elemeinek darabszámát a paraméterlistán, a páros elemeinek darabszámát pedig függvényértékként adja vissza. Az *AtlagSzamol()* függvény a tömb adatainak átlagát számítja ki, és visszaadja a paraméterlistán. Írjunk hozzá ***main()*** függvényt! Aktiváljuk a függvényeket a ***main()*** függvényben, és jelenítsük meg az eredményeket! (GYAK6_2)

A program futásának eredménye:

```
Elemek szama (max. 50): 5
0. adat: 3
1. adat: -2
2. adat: 0
3. adat: 6
4. adat: 4

Pozitiv adatok szama   : 3
Negativ adatok szama   : 1
Zerus adatok szama     : 1
Paros adatok szama     : 4
Paratlan adatok szama : 1

Adatok osszge   : 11
Adatok atlaga   : 2.200000
```

3. Használjuk az alábbi definíciókat és deklarációkat!

```
void KoordOlvas(int *x, int *y);
int Ternegyed(int x, int y);
void TernegyedKiir(int k);
int main()
{
  int xx, yy, t;
  . . .
}
```

Írjunk *KoordOlvas()* függvényt, amely beolvassa egy pont koordinátáit! A *Ternegyed()* függvény a koordinátapont egész számmal jelzett térnegyedet függvényértékként adja vissza ((+,+) 1, (-,+) 2, (-,-) 3, (+,-) 4, (0,0) origó 0). A *TernegyedKiir()* függvény bemenő paraméterként kapja a térnegyed számát, és szövegesen írja vissza a térnegyedet! Mindhárom függvényt aktiváljuk a ***main()*** függvényben! (GYAK6_3)

A program futásainak eredménye:

```
x koordinata: 2
y koordinata: 3
+,+ terfel

x koordinata: 0
y koordinata: 0
0,0 origo
```

4. Használjuk fel az alábbi prototípusokat, és definíciókat!

```
int* Olvas(int *n);
int Szamol(int *y, int n, int *ossz);
int main()
{
  int *x, db, szorzat, osszeg;
  . . .
}
```

Az *Olvas()* függvényben kérdezzünk rá az aktuális adatok számára *(*n)*, dinamikusan foglaljuk le a tömb számára a memóriaterületet, és töltsük fel a tömböt egész típusú adatokkal. Az **int*** mutatót függvényértékként adjuk vissza!

```
int* Olvas(int *n)
{
  int *w, i;
  printf("Adatok szama: "); scanf("%d",n);
  w = (int*)malloc(*n * sizeof(int));
  if (w != NULL)
  {  for (i = 0; i<*n; i++)
     {
      printf("%d. adat: ", i); scanf("%d",&w[i]);
     }
   }
  return w;
}
```

A *Szamol()* függvényben az *y* tömb és az *n* elemszám ismeretében számítsuk ki a tömb páros elemeinek összegét, melyet a paraméterlistán és a páratlan indexű elemeinek szorzatát, melyet függvényértékként adunk vissza!

Írjuk meg a függvényeket, és tervezzünk hozzá ***main()*** függvényt, aktiváljuk a függvényeket, majd írjuk ki a *szorzat* és az *osszeg* eredményt! (GYAK6_4)

A program futásának eredménye:

```
Adatok szama: 5
0. adat: 1
1. adat: 2
2. adat: 3
3. adat: 4
4. adat: 5

Szorzat : 8
Osszeg  : 9
```

5. Használjuk fel az alábbi prototípusokat és definíciókat!

```
typedef struct{
                  double alap, hatvany;
                  int kitevo;
                } hatv;
hatv Olvas(void);
```

```
hatv HatvanySzamol(hatv h);

int main()
{ hatv v;
  . . .
}
```

A *hatv* felhasználói típus olyan struktúrát jelöl, amelynek valós *alap* és *hatvány*, valamint egész *kitevo* adattagja van.

Az *Olvas()* függvényben értéket adunk a *hatv* típusú struktúra *alap* és *hatvány* adattagjainak, és függvényértékként adjuk vissza a struktúrát!

A *HatvanySzamol()* függvényben kiszámítjuk a bemenő paraméterként kapott *hatv* típusú struktúra *hatvany* értékét, és a módosított struktúrát függvényértékként adjuk vissza.

Írjuk meg a függvényeket, tervezzünk hozzájuk ***main()*** függvényt, majd jelenítsük meg az eredményeket! (GYAK6_5)

A program futásának eredménye:

```
alap  : 2
kitevo: 3

Hatvany : 8.000000
```

6. Használjuk fel az alábbi prototípusokat és definíciókat!

```
#define PI 3.141592654
typedef struct {
                  double r, felsz;
                 }Gomb;
Gomb *Olvas(int *db);
void KeresMinFelsz(Gomb *b, int db, double *MFelsz);
int main()
{
  Gomb *g;
  int g_db;
  double minfelsz;
```

Az *Olvas()* függvényben olvassuk be a struktúratömb elemeinek darabszámát, melyet a paraméterlistán adunk vissza! Foglaljunk helyet a dinamikus struktúratömbnek, adjunk értéket a *r* adattagnak, és számítsuk ki a gömb felületét, majd függvényértékként adjuk vissza a struktúratömb kezdőcímét.

```
Gomb *Olvas(int *db)
{
   Gomb *b = NULL;
   int i;
   printf("Adatok szama: "); scanf("%d",db);
   b = (Gomb*)malloc(*db *sizeof(Gomb));
   if (b != NULL)
   {
     for (i = 0; i<*db; i++)
     {
        printf("%d. sugar: ",i); scanf("%lf",&b[i].r);
        b[i].felsz = 4 * pow(b[i].r, 2) * PI;
```

```
            printf("              felszin: %lf\n",b[i].felsz);
        }
    }
    return b;
}
```

A *KeresMinFelsz()* függvényben keressük meg a gömbök közül a legkisebb felszínűt, és adjuk vissza az *MFelsz* paraméterben!

Írjuk meg a függvényeket, és tervezzünk hozzájuk ***main()*** függvényt, írassuk ki az eredményeket! (GYAK6_6)

7. Készítsünk programot, amely a beírt karaktersorozatot egy minta alapján ellenőrzi. A minta szintén sztring, melynek karakterei az ellenőrizendő karaktersorozat azonos pozíciójában álló karakterek milyenségét definiálják: A - nagybetű, a - kisbetű, 0 - decimális jegy. Más karakter esetén az adott karakter megléte szükséges, míg a kérdőjel tetszőleges karaktert engedélyez az adott pozícióban. Például a rendszám ellenőrzéshez használható mintasztring: "*AAA-000*" (GYAK6_7).

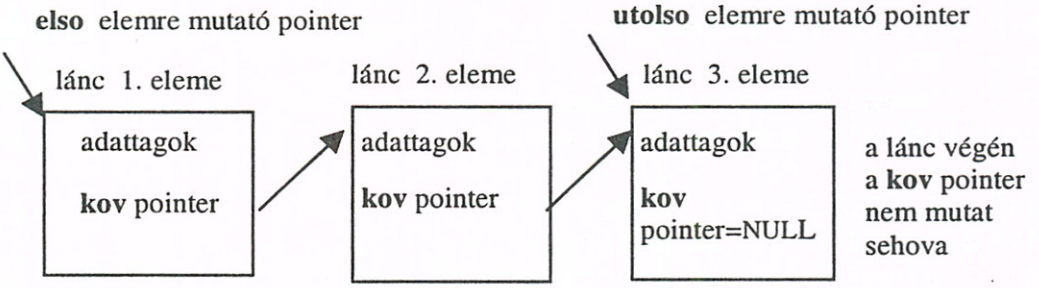
A program futásának eredményei:

```
Kerek egy rendszamot (AAA-000): DCA-036
A DCA-036 rendszam.

Kerek egy telefonszamot: (000-00-00) 456-1231
A 456-1231 nem telefonszam.
```

7. Gyakorlat

Rövid elméleti összefoglaló

- Egyirányú láncolt lista. Az láncelem az adatok mellett a következő elemre mutató pointert is tárolja (*kov*). A mintafeladatban a *kov* pointer *Doboz* típusú struktúrára tud mutatni. (*MINTA7_01*)



A *Doboz* struktúra adattagjai:

- *a,b,c* a doboz (téglatest) három oldala,
- *terf* a doboz térfogata.

DobozOlvas() adott számú doboz adatait olvassa be, és egyirányú láncba fűzi. Az első eleme mutató pointert adja vissza.
Lista() kiírja a láncolt lista tartalmát.
KeresMaxDoboz() megkeresi a maximális térfogatú dobozt, és kiírja az adataival együtt.

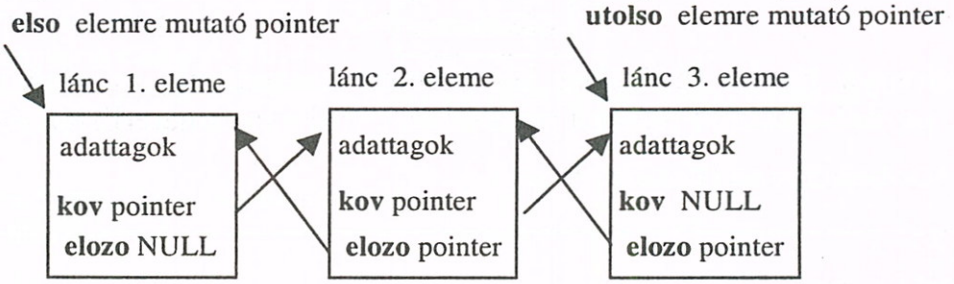
```
/*MINTA7_01*/
#include <stdio.h>
#include <process.h>  /* az exit() használatához */
typedef struct d {
                    double a,b,c;  /* a doboz oldalai */
                    double terf;   /* a doboz térfogata */
                    struct d kov;  /* következő dobozra mutató
                                      pointer */
                  } Doboz;
Doboz *DobozOlvas();
void Lista(Doboz *p);
void KeresMaxDoboz(Doboz *p);
int main()
{ Doboz *w_elso, *p, *p1;
  w_elso = DobozOlvas(); /* dobozok beolvasása és láncba fűzése */
  Lista(w_elso);         /* lista elemeinek kiíratása */
  KeresMaxDoboz(w_elso); /* max. térfogatú doboz keresése */
  p = w_elso;
  while (p != NULL) /* a láncolt lista által lefoglalt memória-
                       terület felszabadítása */
  {
    p1 = p->kov;
    free(p); p = p1;
  }
  w_elso = NULL;
  fflush(stdin); getchar();
  return 0;
}
```

```
Doboz *DobozOlvas(void)
{ int i, n;
  Doboz *uj, *elso = NULL, *utolso = NULL;
  printf("Dobozok szama: ");
  scanf("%d",&n);  /* dobozok darabszámának beolvasása */
  for (i = 0; i<n; i++)
  {
    uj = (Doboz*)malloc(sizeof(Doboz));/* memóriaterület lefoglalása */
    if (uj == NULL)
    { printf("Memoriafoglalas nem sikerult!\n");
      fflush(stdin); getchar(); exit(1);
    }
    printf("\nA oldal: "); scanf("%lf",&uj->a);
    printf("B oldal: "); scanf("%lf",&uj->b);
    printf("C oldal: "); scanf("%lf",&uj->c);
    uj->terf = uj->a * uj->b * uj->c;
    uj->kov = NULL;
    if (elso == NULL) elso = uj;
    else
    { utolso->kov = uj; }
    utolso = uj;
  }
  return elso;
}
void Lista(Doboz *p)
{ if (p != NULL)
  {
    printf("\n A oldal   B oldal   C oldal  Terfogat\n");
    while (p != NULL)
    {
      printf("%8.2lf  %8.2lf  %8.2lf  %8.2lf\n",
             p->a, p->b, p->c, p->terf);
      p = p->kov;
    }
  }
  else printf("A lanc ures!\n");
}
void KeresMaxDoboz(Doboz *p)
{  Doboz max; max.terf = 0;
   if (p != NULL)
   {
    while (p != NULL) /* legnagyobb térfogatú doboz keresése */
    {
      if (p->terf > max.terf) max = *p;  p = p->kov;
    }
    if (max.terf > 0)
    {
      printf("\n A oldal   B oldal   C oldal  Max terfogat\n");
      printf("%8.2lf  %8.2lf  %8.2lf  %8.2lf\n",
             max.a, max.b, max.c, max.terf);
    }
  }
}
```

A program futásának eredménye:

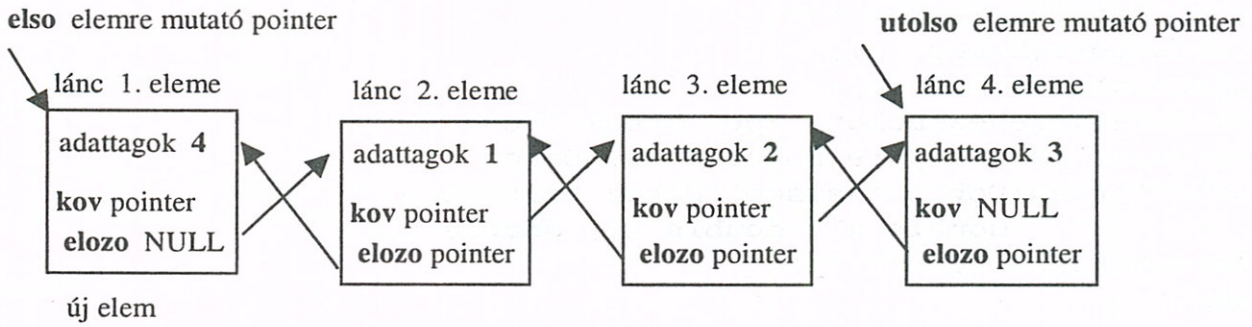
```
Dobozok szama: 3

A oldal: 2
B oldal: 1
C oldal: 3

A oldal: 4
B oldal: 5
C oldal: 6

A oldal: 2
B oldal: 7
C oldal: 4

 A oldal    B oldal    C oldal   Terfogat
    2.00       1.00       3.00       6.00
    4.00       5.00       6.00     120.00
    2.00       7.00       4.00      56.00

 A oldal    B oldal    C oldal   Max terfogat
    4.00       5.00       6.00     120.00
```

- **Kétirányú láncolt lista.** A lánc elemei nem csak a következő elemet ismerik (*kov*), hanem az előzőt (*elozo*) is. A listaelem törlésének bemutatása. (*MINTA7_02*)

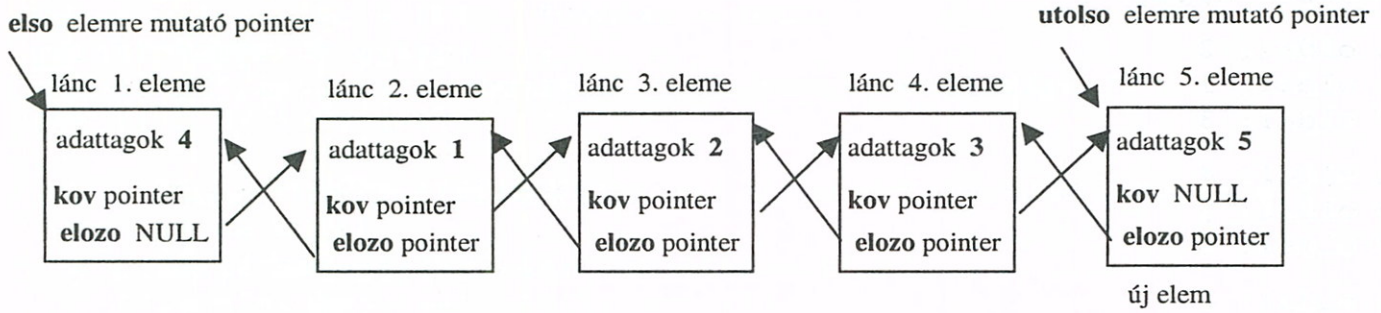


Természetesen vizsgálatot kell végezni, hogy a törlendő elem a lánc elején vagy a végén helyezkedik-e el, vagy a lánc közbülső eleme.

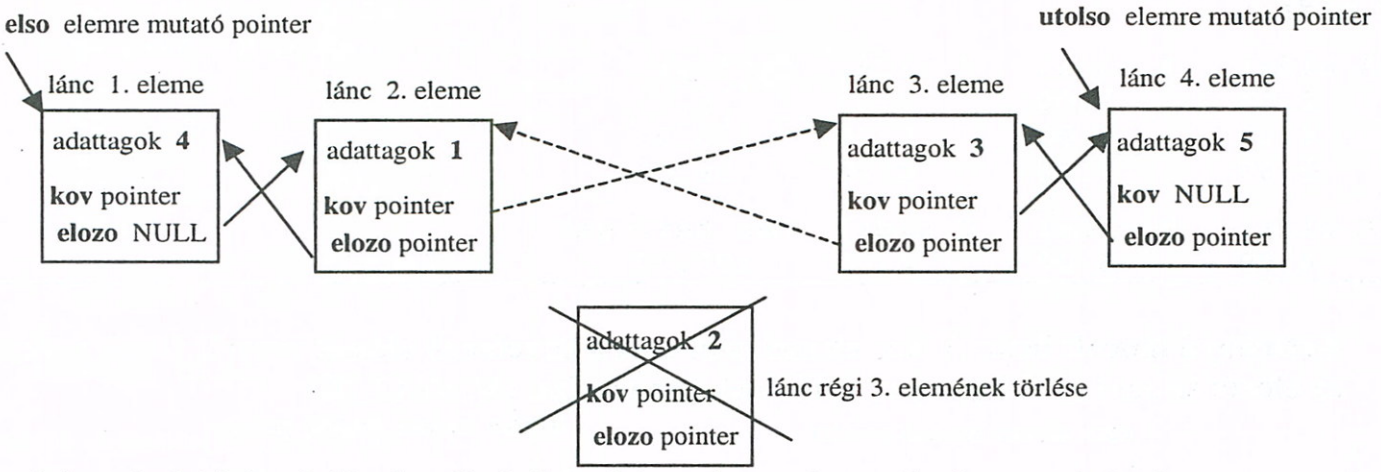
Hozzáadás a lánc elejére:



Hozzáadás a lánc végére:



Törlés a lánc közepéből:



A dobozok kétirányú láncba fűzéséhez két pointert használunk, az első elemre mutatót a **return** mellett, az utolsó elemre mutatót pointerben (**p) a paraméterlistán adjuk vissza:

```
Doboz *DobozOlvas(Doboz **p);
```

```
/*MINTA7_02*/
#include <stdio.h>
#include <process.h>
typedef struct d {
                  double a,b,c;  /* a doboz oldalai */
                  double terf;   /* a doboz térfogata */
                  struct d *elozo;
                  struct d *kov;
                 } Doboz;
 Doboz *DobozOlvas(Doboz **p);
void ListaElore(Doboz *p);
void ListaHatra(Doboz *p);
void BeszurasVegehez(Doboz **pe, Doboz **pu);
void BeszurasListaElejere(Doboz **pe, Doboz **pu);
void DobozTorles(Doboz **elsop, Doboz **utolsop,
                 double x1, double x2, double x3);
void KeresMaxDoboz(Doboz *p);
int main()
{ Doboz *w_elso = NULL, *w_utolso = NULL, *p, *p1;
  double aa, bb, cc;
  w_elso = DobozOlvas(&w_utolso);
  ListaElore(w_elso);
  BeszurasVegehez(&w_elso, &w_utolso);
```

```
  ListaElore(w_elso);
  BeszurasListaElejere(&w_elso, &w_utolso);
  ListaElore(w_elso);
  printf("\nDoboz adatai a torleshez:");
  printf("\nA oldal: "); scanf("%lf",&aa);
  printf("B oldal: "); scanf("%lf",&bb);
  printf("C oldal: "); scanf("%lf",&cc);
  DobozTorles(&w_elso, &w_utolso,aa,bb,cc);
  ListaHatra(w_utolso);
  KeresMaxDoboz(w_elso);
  p = w_elso;
  while (p != NULL)
  {
    p1 = p->kov;
    free(p);
    p = p1;
  }
  w_elso = NULL;
  w_utolso = NULL;
  fflush(stdin);getchar();
  return 0;
}
Doboz *DobozOlvas(Doboz **p)
{ int i, n;
  Doboz *uj, *elso = NULL, *utolso = NULL;
  printf("Dobozok szama: ");
  scanf("%d",&n);  /* a dobozok darabszámának beolvasása */
  for (i = 0; i<n; i++)
  {
    uj = (Doboz*)malloc(sizeof(Doboz));
    if (uj == NULL)
    {
      printf("Memoriafoglalas nem sikerult!\n");
      fflush(stdin); getchar(); exit(1);
    }
    printf("\nA oldal: "); scanf("%lf",&uj->a);
    printf("B oldal: "); scanf("%lf",&uj->b);
    printf("C oldal: "); scanf("%lf",&uj->c);
    uj->terf = uj->a * uj->b * uj->c;
    uj->kov = NULL;
    uj->elozo = NULL;
    if (elso == NULL) elso = uj;
    else
    {
      utolso->kov = uj;
      uj->elozo = utolso;
    }
    utolso = uj;
  }
  *p = utolso;  /* a paraméterlistán az utolsó elemre mutató */
  return elso;  /* az első elemre mutató */
}
```

```
void ListaHatra(Doboz *p)
{
  if (p != NULL)
  {
    printf("\nListazas visszafele");
    printf("\n A oldal   B oldal   C oldal  Terfogat\n");
    while (p != NULL)
    {
      printf("%8.2lf  %8.2lf  %8.2lf  %8.2lf\n",
             p->a, p->b, p->c, p->terf);
      p = p->elozo;
    }
  }
  else printf("\nA lanc ures\n");
}

void ListaElore(Doboz *p) /* dobozok adatainak megjelenítése előre */
{
  if (p != NULL)
  {
    printf("\nListazas elore");
    printf("\n A oldal   B oldal   C oldal  Terfogat\n");
    while (p != NULL)
    {
      printf("%8.2lf  %8.2lf  %8.2lf  %8.2lf\n",
             p->a, p->b, p->c, p->terf);
      p = p->kov;
    }
  }
  else printf("\nA lanc ures!\n");
}

void KeresMaxDoboz(Doboz *p) /* legnagyobb térfogatú doboz keresése */
{
   Doboz max; max.terf = 0;
   if (p != NULL)
   {
    while (p != NULL)
    {
      if (p->terf > max.terf) max = *p;
      p = p->kov;
    }
    if (max.terf > 0)
    {
      printf("\n A oldal   B oldal   C oldal  Max terfogat\n");
      printf("%8.2lf  %8.2lf  %8.2lf  %8.2lf\n",
             max.a, max.b, max.c, max.terf);
    }
  }
  else printf("\nA lanc ures!\n");
}
```

```
void BeszurasVegehez(Doboz **pe, Doboz **pu)
{
   Doboz *uj, *p;
   p = *pu;
   printf("\nUj doboz beszurasa a lanc vegehez\n");
   uj = (Doboz*)malloc(sizeof(Doboz));
   if (uj == NULL)
    {
      printf("Memoriafoglalas nem sikerult!\n");
      fflush(stdin); getchar(); exit(1);
    }
   printf("A oldal: "); scanf("%lf",&uj->a);
   printf("B oldal: "); scanf("%lf",&uj->b);
   printf("C oldal: "); scanf("%lf",&uj->c);
   uj->terf = uj->a * uj->b * uj->c;
   uj->elozo = NULL;
   uj->kov = NULL;
   if (p != NULL)
   {
     p->kov = uj;
     uj->elozo = p;
     *pu = uj;
   }
   else
   {
     *pe = uj;
     *pu = uj;
   }
}
void BeszurasListaElejere(Doboz **pe, Doboz **pu)
{
   Doboz *uj, *p;
   p = *pe;
   printf("\nUj doboz beszurasa a lanc elejere\n");
   uj = (Doboz*)malloc(sizeof(Doboz));
   if (uj == NULL)
    {
      printf("Memoriafoglalas nem sikerult!\n");
      fflush(stdin); getchar(); exit(1);
    }
   printf("A oldal: "); scanf("%lf",&uj->a);
   printf("B oldal: "); scanf("%lf",&uj->b);
   printf("C oldal: "); scanf("%lf",&uj->c);
   uj->terf = uj->a * uj->b * uj->c;
   uj->elozo = NULL;
   uj->kov = NULL;
   if (p != NULL)
   {
     uj->kov = p;
     p->elozo = uj;
     *pe = uj;
   }
```

```
    else
    {
      *pu = uj;  *pe = uj;
    }
}
void DobozTorles(Doboz **elsop, Doboz **utolsop,
                 double x1, double x2, double x3)
{
  Doboz *p, *p1, *p2;
  int torolve = 0;
  p = *elsop;
  while (p != NULL)
  {
    if (p->a == x1 && p->b == x2 && p->c == x3)
    {
      p1 = p->elozo;
      p2 = p->kov;
      if (p1 == NULL)
      {
         *elsop = p2;
         if (p2 != NULL) p2->elozo = NULL;
         else *utolsop = p2;
      }
      else
      {
         p1->kov = p2;
         if (p2 != NULL)
         {
           p2->elozo = p1;
         }
         else *utolsop = p1;
      }
      p->kov = NULL;
      p->elozo = NULL;
      torolve = 1;
      printf("Doboz torolve.\n");
      free(p);
      break;
    }
    p = p->kov;
  }
  if (torolve == 0)
    printf("Doboz torlese nem sikerult.\n");
}
```

Megjegyzések a függvényekhez:

A *DobozOlvas()* adott számú dobozt beolvas, lefoglalva számukra a memóriaterülete és kétirányú láncba fűzi őket.

A *ListaHátra()* kiírja a láncolt lista tartalmát az utolsó elemtől kezdve visszafelé.

A *ListaElore()* kiírja a láncolt lista tartalmát az első elemtő kezdve előre.

A *BeszurasVegehez()* pointer paramétereiben adja vissza az első elemre (pe) és az utolsó elemre (pu) mutató pointerek változásait.

- Ha az utolsó elemre mutató (**pu*) pointer nem *NULL*, akkor a láncnak már van eleme. Lánc utolsó elemének *kov* pointere átveszi az *uj* doboz címét, és az *uj* láncelem *elozo* pointere pedig a régi utolsó elemre mutat, majd az utolsó (*pu) az *uj* láncelemre mutat.
- Ha az utolsó pointer (**pu*) értéke *NULL*, akkor az *uj* láncelem első és egyben az utolsó eleme lesz a láncnak, ezért az első (**pe*) és az utolsó (**pu*) pointer is rámutat.

A *BeszurasListaElejére()* pointer paramétereiben veszi át és adja vissza az első elemre (pe) és az utolsó elemre (pu) mutató pointerek változásait.

- Ha az első (**pe*) elemre mutató pointer nem *NULL*, akkor a láncnak már van eleme. Az *uj* elem *kov* pointere a lánc első elemének címét veszi át, és a lánc első elemének *elozo* eleme pedig az új elemre mutat.
- Ha az első (**pe*) pointer értéke *NULL*, akkor az *uj* láncelem első és egyben az utolsó eleme lesz a láncnak, ezért az első (**pe*) és az utolsó (**pu*) pointer is rámutat.

A *DobozTorles()* függvény pointer paramétereiben veszi át, és adja vissza az első elemre (*pe*) és az utolsó elemre (*pu*) mutató pointerek változásait, valamint a doboz három oldalának hosszát, melyet törölni kell a láncból. A *torolve* változó nullázzuk, ebben fogjuk jelezni, hogy megtörtént a törlés (1).

Az *x1, x2* és *x3* adatot megkeresi láncban, és ha megtalálta, törli a lánc elemét. Természetesen figyelni kell arra, hogy a törlendő elem a lánc elején, a lánc belsejében vagy a végén van-e. A lánc végét a *NULL* pointer jelzi.

- A *p* pointerbe felvesszük a lánc első elemére mutató pointert (**elsop*). A **while** ciklus feltétele ellenőrzi, hogy van-e még eleme a láncnak, ekkor a *p* értéke nem *NULL*. A lánc végét a *NULL* jelzi.
- A megtalált elemnél *p1* változóba tároljuk a *p* elem *elozo*, a *p2* változóba pedig a *p* elem *kov* pointerét.
- Ha a *p1*, azaz a *p elozo* pointere *NULL*, akkor nincs *elozo* eleme, tehát ő az első elem, ezért az *elso (*elsop)* átveszi a *p2*, azaz *p* következő láncra mutató címét. Ha a *p2* nem NULL, az *elozo* pointerét nullázzuk, ha értéke NULL, akkor az utolsó (**utolsop*) is átveszi a *p2* pointert.
- Ha nem első elem, akkor a *p elozo* elemének (*p1*) a *kov* tagjába a *p* következő elemének a címét (*p2*) írjuk bele. Ha a *p2* nem *NULL*, ez azt jelenti, hogy a lánc belsejében vagyunk, ezért a *p* következő elemének (*p2*) az *elozo* tagját írjuk a *p* előző elemre mutató címét (*p1*).
- Ha a *p2 NULL* volt, amely azt jelenti, hogy a törlendő elem az utolsó elem volt, ezért a *p1* lesz az *utolso* láncelem (**utolsop*), amire a *p1* visszamutatott.
- Nullázzuk a *p elozo* és *kov* tagját, és a ***free()*** függvénnyel töröljük a *p* elemet, a *torolve* változót pedig 1-re állítjuk.

Ha a *torolve* változó értéke zérus maradt, akkor a "*Doboz torlese nem sikerult!*" szöveg jelenik meg.

A *MINTA7_02* mintafeladatban a főfüggvény bemutatja a *DobozOlvas()* függvényt, amely adott számú dobozt fűz láncba. A *BeszurasElejere()* függvény az új elemet a lánc elejére, a *BeszurasVegehez()* függvény a lánc végére fűzi. A *ListaElore()* és *ListaHatra()* függvényekkel ellenőrizzük a lánc tartalmát. A *DobozTorles ()* függvénnyel törlünk a láncból.

A program futásának eredménye:

```
Dobozok szama: 1

A oldal: 1
B oldal: 1
C oldal: 1

Listazas elore
 A oldal   B oldal   C oldal  Terfogat
    1.00      1.00      1.00      1.00

Uj doboz beszurasa a lanc vegehez
A oldal: 2
B oldal: 2
C oldal: 2

Listazas elore
 A oldal   B oldal   C oldal  Terfogat
    1.00      1.00      1.00      1.00
    2.00      2.00      2.00      8.00
Uj doboz beszurasa a lanc elejere
A oldal: 3
B oldal: 3
C oldal: 3

Listazas elore
 A oldal   B oldal   C oldal  Terfogat
    3.00      3.00      3.00     27.00
    1.00      1.00      1.00      1.00
    2.00      2.00      2.00      8.00

Doboz adatai a torleshez:
A oldal: 2
B oldal: 2
C oldal: 2
Doboz torolve.

Listazas visszafele
 A oldal   B oldal   C oldal  Terfogat
    1.00      1.00      1.00      1.00
    3.00      3.00      3.00     27.00

 A oldal   B oldal   C oldal  Max terfogat
    3.00      3.00      3.00     27.00
```

Feladatok

1. Használjuk az alábbi definíciókat! (GYAK7_1)

```
typedef struct AruElem{
                        char azonosito[20];
                        int db;
                        double ara;
                        struct AruElem *kov;
                       } Aru;
Aru *AruOlvas();
```

```
void AruLista(Aru *p);
double OsszAr(Aru *p);
int OsszDb(Aru *p);
int main()
{
  Aru *elsoAru, *p, *p1;
  double RaktarErteke = 0;
  int RaktarArukDb = 0;
  . . .
}
```

Az *Aru* struktúra az áru azonosítóját, darabszámát és az árát tartalmazza.

Az *AruOlvas()* függvényben olvassuk be a raktár tartalmát, fűzzük egyirányú láncba, és függvényértékként adjuk vissza a lánc első elemére mutató pointert!

Az *OsszAr()* függvényben számítsuk ki a raktár értékét, azaz számítsuk ki az áruk ára és daraszámuk szorzatösszegét, melyet függvényértékként adjunk vissza!

Az *OsszDb()* függvényben számláljuk meg a raktárban lévő áruk darabszámát, melyet függvényértékként adjunk vissza!

Az *AruLista()* függvénnyel jelenítsük meg a láncelemek adattagjait!

A program futásának eredménye:

```
Arufajtak szama: 3

Aru azonsitoja: asztal
Darab szama   : 5
Ara           : 1000

Aru azonsitoja: szek
Darab szama   : 3
Ara           : 500

Aru azonsitoja: szekreny
Darab szama   : 1
Ara           : 12000

Azonosito       Darab    Ara
-------------------------------------
asztal          5          1000.00
szek            3           500.00
szekreny        1         12000.00

A raktar erteke: 18500.00
A raktar darab tartalma: 9
```

2. Használjuk az alábbi definíciókat! (GYAK7_2)

```
#include <stdio.h>

#include <math.h>
#define PI 3.141592654
```

```
typedef struct ngy {
                    double a;
                    double sugar;
                    double korter;
                    struct ngy *elozo;
                    struct ngy *kov;
                   } Negyzet;

Negyzet *NegyzetOlvas(Negyzet **p);
void ListaElore(Negyzet *p);
void KorTerSzamitas(Negyzet *p);
double OsszesKorTer(Negyzet *p);
int main()
{
  Negyzet *n_elso = NULL, *n_utolso = NULL, *p, *p1;
  double OsszTer;
  . . .
}
```

A *Negyzet* struktúra adattagjai: *a* a négyzet oldala, *r* a négyzetbe írható kör sugara, *korter* a kör területe.

A *NegyzetOlvas()* függvény olvassa be az adott számú négyzet struktúra adatait, fűzze kétirányú láncba, és a lánc első elemére mutató pointert a függvény visszatérési értékként, a lánc utolsó elemére mutató pointert pedig a paraméterlistán (*p* mutató) adja vissza!

A *KorTerSzamitas()* függvény számítsa ki a lánc összes elemének a négyzetbe írható kör területét!

Az *OsszesKorTer()* függvény összegezze a lánc minden elemének *korter* adattagját!

A program futásának eredménye:

```
Negyzetek szama: 3

Negyzet oldala: 2
Negyzet oldala: 5
Negyzet oldala: 7

   A oldal    Sugar    Kor terulet
---------------------------------------
     2.00      1.00       3.14
     5.00      2.50      19.63
     7.00      3.50      38.48

A korteruletek osszege:    61.26
```

8. Gyakorlat

Rövid elméleti összefoglaló

- **Állományok típusai**

  Szöveges állományok

  A szöveges állomány tetszőleges hosszúságú sorokból épül fel, a sor végét '\n' zárja. A '\n' karakter értelmezése az operációs rendszertől függ. A Windows rendszereknél, szöveges állományok esetében a programbeli '\n' karakter a CR/LF (kocsi vissza, soremelés) karaktereknek felel meg, a Unix (Linux) operációs rendszerek esetében pedig az LF karakternek. A szöveges állományok kezelésnél a '\n' karakter automatikusan konvertálódik az operációs rendszernek megfelelő fizikai reprezentációba. Az ún. bináris módban megnyitott adatfolyamoknál ez a konverzió elmarad. Az állomány végét az EOF (**E**nd **O**f **F**ile) vezérlőkarakter zárhatja.

  Bináris állományok

  A bináris állományok bájtokból épülnek fel.

- **Állományok kezelésének általános leírása**

  Az állományok tartalmának eléréséhez mindig ugyanazokat a lépéseket kell végrehajtani:
  - a fájlt azonosító mutató definiálása,
  - az állomány megnyitása,
  - az állomány tartalmának feldolgozása fájlműveletek (olvasás, írás, pozicionálás stb.) felhasználásával.
  - A műveletek elvégzése után az állomány lezárása.

  Fájlmutató definiálása

  A programban a felhasználni kívánt adatállományhoz külön fájlmutatót kell létrehoznunk. A ***FILE**** típusú fájlmutató – melynek deklarációja az *STDIO.H* állományban található – azonosítja az állományt.

  Az állomány lezárása

  Az írásra és olvasásra megnyitott állományokat a használat végén az ***fclose()*** függvény hívásával le kell zárnunk.

  Ha szöveges állományon írási műveletet hajtottunk létre, akkor az ***fflush()*** függvénnyel a memóriában található átmeneti pufferek tartalmát lezárás előtt az állományba kell írnunk.

  Mindkét függvénynek argumentuma a fájlmutató:

```
FILE *f;
  . . .
fflush(f);
fclose(f);
```

  Az állomány végének vizsgálata

  Az állomány olvasásakor az ***feof()*** függvény 1 értéke jelzi az állományvége eseményt.

```
int feof(FILE *f);
```

Az állomány végének figyelését általában **while** ciklusban végezzük:

```
while (!feof(f))
{
   /* az állomány olvasása */
}
fclose(f);
```

- **Szöveges állomány használata**

A szöveges állomány megnyitása

A háttértáron lévő állomány tartalmához csak a fájl megnyitása után férhetünk hozzá. A megnyitás hozza létre a kapcsolatot a fájlmutató és az operációs rendszer állománya között. A nyitáskor meg kell adnunk az állomány típusát és a hozzáférés módját:

```
FILE *fopen(const char *fajlnev, const char *mod);
```

A *fajlnev* paraméter helyén definiáljuk az állomány operációs rendszerben használt nevét, melyben elérési útvonal is szerepelhet.
A *mod* paraméter szintén sztring, amely a fájl elérését és típusát határozza meg:

Szöveges	*A **mod** paraméter jelentése*
`"r", "rt"`	Létező állomány megnyitása.
`"w", "wt"`	Új állomány megnyitása. Ha már létezik, akkor a tartalma elvész.
`"a", "at"`	Az állomány megnyitása a végéhez való hozzáírásra, tehát a fájl bővítése. Ha az állomány nem létezik, akkor nyitáskor létrejön.
`"r+", "r+t"`	A létező állomány megnyitása írásra és olvasásra (*update*).
`"w+", "w+t"`	Új állomány megnyitása írásra és olvasásra (*update*). Ha a fájl már létezik, akkor a tartalma elvész.
`"a+", "a+t"`	Az állomány megnyitása a végén végzett írásra és olvasásra (*update*). Ha a fájl nem létezik, akkor most jön létre.

Ha nem sikerült az állomány megnyitása, akkor adjunk hibajelzést!

```
if ((f1 = fopen("mondat.txt","r")) == NULL)
{
   fprintf(stderr,"A mondat.txt allomany nem letezik!\n");
   ...
}
```

Karakter olvasása és írása

Karakter olvasása:

```
int fgetc(FILE *adatfolyam);
```

Karakter írása:

```
int fputc(int c, FILE *adatfolyam);
```

A *MINTA8_01* mintafeladat a *MONDAT.TXT* állományból karakterenként olvas, megjeleníti a beolvasott karaktereket, melyeket azután nagybetűssé alakítva az *UJMONDAT.TXT* állományba ír.

```
/*MINTA8_01*/
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

int main()
{
  FILE *f1, *f2;
  int c, c2;
  if ((f1 = fopen("mondat.txt","r")) == NULL)
  {
    fprintf(stderr,"A mondat.txt allomany nem letezik!\n");
    getchar();
  }
  else
  {
    f2 = fopen("ujmondat.txt","w");
    printf("mondat.txt tartalma: \n");
    c = ' ';
    while (c != '!' && !feof(f1))
    {
      c = fgetc(f1);
      printf("%c",c); c2 = toupper(c);
      fputc(c2,f2);
    }
    printf("\n");
    fclose(f1);
    fclose(f2);
  }
  getchar();
  return 0;
}
```

A program futásának eredménye:	A *ujmondat.txt* állomány tartalma:
`mondat.txt tartalma:` `Programozzunk C nyelven!`	PROGRAMOZZUNK C NYELVEN!

Szóközt tartalmazó karaktersorozat olvasása és írása

Egy sort olvas, amely szóközt is tartalmazhat:

```
char *fgets(char *s, int n, FILE *adatfolyam);
```

Egy sort ír, amely szóközt is tartalmazhat:

```
int fputs(const char *s, FILE * adatfolyam);
```

A *MINTA8_02* mintafeladat soronként olvas a *GYUMOLCS.TXT* állományból, a megjelenítésnél azonban csak a páratlan sorszámú szöveget sorszámozza meg, és a beolvasott szöveget kisbetűssé alakítva a *KGYUMOLCS.TXT* állományba írja.

```
/*MINTA8_02*/
#include <stdio.h>
#include <ctype.h>
int main()
{
  FILE *f1, *f2;
  char gyum[40], gyum2[40];
  int i, sorszam = 0;
  if ((f1 = fopen("gyumolcs.txt","r")) == NULL)
  {
    fprintf(stderr,"A gyumolcs.txt allomany nem letezik!\n");
    getchar();
  }
  else
  {
    printf("gyumolcs.txt tartalma: \n\n");
    f2 = fopen("Kgyumolcs.txt","w");
    while (!feof(f1))
    {
      fgets(gyum,40,f1);
      sorszam++;
      if (sorszam % 2 != 0)
         printf("%2d %s",sorszam,gyum);
      else
         printf("   %s",gyum);
      for (i = 0; gyum[i]; i++)
        gyum2[i] = tolower(gyum[i]); /* kisbetűvé alakít */
      gyum2[i]= '\0';
      fputs(gyum2, f2);
    }
    printf("\n");
    fclose(f1); fclose(f2);
  }
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
gyumolcs.txt tartalma:

 1 ALMA
   KORTE
 3 DIO
   ANANASZ
 5 SZILVA
```

Kgyumolcs.txt tartalma:

```
alma
korte
dio
ananasz
szilva
szilva
```

Formázott adatok olvasása és írása

A *scanf()* és *printf()* függvényekhez hasonlóan formázott adatokat olvashatunk állományból és írhatunk állományba:

```
int fscanf(FILE * adatfolyam, const char *formatum, ...);
int fprintf(FILE * adatfolyam, const char *formatum, ...);
```

A *MINTA8_03* mintafeladat megadott darabszámú *Doboz* típusú struktúra adattagjait formázva írja ki a *DOBOZOK.TXT* állományba.

```
/*MINTA8_03*/
#include <stdio.h>

typedef struct d {
                   double a,b,c;  /* a doboz oldalai */
                   double terf;
                   struct d *elozo;
                   struct d *kov;
                  } Doboz;

int main()
{ int i,n;
  Doboz uj;
  FILE *fd;
  fd = fopen("dobozok.txt","w");
  printf("dobozok.txt allomany irasa\n");
  printf("Dobozok szama: ");
  scanf("%d",&n);
  for (i = 0; i<n; i++)
  {
    printf("\nA oldal: "); scanf("%lf",&uj.a);
    printf("B oldal: "); scanf("%lf",&uj.b);
    printf("C oldal: "); scanf("%lf",&uj.c);
    uj.terf = uj.a * uj.b * uj.c;
    uj.kov = NULL;
    uj.elozo = NULL;
    fprintf(fd,"%8.2lf %8.2lf %8.2lf %8.2lf\n",
              uj.a,uj.b,uj.c,uj.terf);
  }
  fclose(fd);
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
dobozok.txt allomany irasa
Dobozok szama: 3

A oldal: 1
B oldal: 1
C oldal: 1
```

```
A oldal: 1
B oldal: 2
C oldal: 3

A oldal: 4
B oldal: 5
C oldal: 6
```

A *MINTA8_04* mintafeladat függvényei:
DobozFOlvas() beolvassa a *DOBOZOK.TXT* állományban tárolt *adatokat* és láncba fűzi.
ListaElore() függvény a lánc elemeit jeleníti meg.
KeresMaxDoboz() megkeresi a legnagyobb térfogatú dobozt, és az adataival együtt megjeleníti azt.

```
/*MINTA8_04*/
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
typedef struct d {
                   double a,b,c;  /* a doboz oldalai */
                   double terf;
                   struct d *elozo;
                   struct d *kov;
                  } Doboz;
Doboz *DobozFOlvas(Doboz **p);
void ListaElore(Doboz *p);
void KeresMaxDoboz(Doboz *p);
int main()
{
  Doboz *w_elso = NULL, *w_utolso = NULL, *p, *p1;
  double aa, bb, cc;
  w_elso = DobozFOlvas(&w_utolso);
  ListaElore(w_elso);
  KeresMaxDoboz(w_elso);
  p = w_elso;
  while (p != NULL)
  {
    p1 = p->kov;
    free(p); p = p1;
  }
  w_elso = NULL; w_utolso = NULL;
  getchar();
  return 0;
}
Doboz *DobozFOlvas(Doboz **p)
{
  Doboz z, *uj, *elso = NULL, *utolso = NULL;
  FILE *fd;
  if ((fd = fopen("dobozok.txt","r")) == NULL)
  {
    fprintf(stderr,"A dobozok.txt allomany nem letezik!\n");
    getchar();
  }
```

```
  else
  {
   while (fscanf(fd,"%lf %lf %lf %lf\n",&z.a,&z.b,&z.c,&z.terf) != EOF)
   {  uj = (Doboz*)malloc(sizeof(Doboz));
      if (uj == NULL)
      {
        printf("Memoriafoglalas nem sikerult!\n");
        getchar(); exit(1);
      }
      uj->a = z.a;  uj->b = z.b; uj->c = z.c; uj->terf = z.terf;
      uj->kov = NULL;
      uj->elozo = NULL;
      if (elso == NULL) elso = uj;
      else
      {
        utolso->kov = uj; uj->elozo = utolso;
      }
      utolso = uj;
    }
    fclose(fd);
  }
  *p = utolso;
  printf("dobozok.txt beolvasasa befejezodott\n");
  return elso;
}
void ListaElore(Doboz *p)
{
  if (p != NULL)
  {
    printf("\nListazas elore");
    printf("\n A oldal   B oldal   C oldal  Terfogat\n");
    while (p != NULL)
    {
      printf("%8.2lf  %8.2lf  %8.2lf  %8.2lf\n",
             p->a, p->b, p->c, p->terf);
      p = p->kov;
    }
  }
  else printf("\nA lanc ures!\n");
}
void KeresMaxDoboz(Doboz *p)
{
   Doboz max; max.terf = 0;
   if (p != NULL)
   {
    while (p != NULL)
    {
      if (p->terf > max.terf) max = *p;
      p = p->kov;
    }
    if (max.terf > 0)
    {
      printf("\n A oldal   B oldal   C oldal  Max terfogat\n");
```

```
      printf("%8.2lf  %8.2lf  %8.2lf  %8.2lf\n",
             max.a, max.b, max.c, max.terf);
    }
  }
  else printf("\nA lanc ures!\n");
}
```

A program futásának eredménye:

```
dobozok.txt beolvasasa befejezodott

Listazas elore
 A oldal   B oldal   C oldal  Terfogat
    1.00      1.00      1.00      1.00
    1.00      2.00      3.00      6.00
    4.00      5.00      6.00    120.00

 A oldal   B oldal   C oldal  Max terfogat
    4.00      5.00      6.00    120.00
```

A *MINTA8_05* mintaprogram a *KOCSIK.TXT* állományt olvassa be, visszaírja táblázatos formában a tartalmát, megkeresi a legfiatalabb évjáratú kocsi, és megszámlálja azoknak a kocsiknak a számát, amelyek 2000-nél korábbi évjáratúak és 5 ajtósak.

```
/*MINTA8_05*/
#include <stdio.h>
typedef struct {
                 char autotip[10];
                 int evjarat;
                 int ajto;
                }gk;
int main()
{ FILE *fgk;
  gk k;
  int legfiatalabb = 1970;
  int ajtok_szama = 0;
  if ((fgk = fopen("kocsik.txt", "r")) == NULL)
  {
     fprintf(stderr,""kocsik.txt allomany nem letezik!\n");
     getchar();
  }
  else
  { printf("kocsik.txt allomany tartalma: \n\n");
    printf("Autotipus  Evjarat Ajtok szama\n");
    printf("------------------------------\n");
    while (fscanf(fgk,"%10s %d %d\n",k.autotip, &k.evjarat, &k.ajto)
           != EOF)
    { printf("%-10s %6d %6d\n",k.autotip,k.evjarat,k.ajto);
      if (legfiatalabb < k.evjarat) legfiatalabb = k.evjarat;
      if (k.ajto == 5 && k.evjarat < 2000) ajtok_szama++;
    }
    fclose(fgk);
    printf("\nLegfiatalabb kocsi: %d  evjaratu\n", legfiatalabb);
```

```
    printf("2000-nel korabbi evjaratu 5 ajtos kocsi: %d  darab\n",
           ajtok_szama);
  }
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
kocsik.txt allomany tartalma:

Autotipus  Evjarat Ajtok szama
-----------------------------
Skoda         2001      4
Mercedes      1985      5
Trabant       1983      2
Opel          2003      5

Legfiatalabb kocsi: 2003  evjaratu
2000-nel korabbi evjaratu 5 ajtos kocsi: 1  darab
```

Feladatok

1. Olvassunk be egy szövegsort a *ki_szoveg* karaktertömbbe a ***gets()*** függvénnyel, és írjuk ki a SZOVEG.TXT állományba! Nyissuk meg az állományt olvasásra és olvassuk be a szövegsort a *be_szoveg* karaktertömbbe! Számláljuk meg a szövegben található *a*, *e* magánhangzókat, és a szóközöket, valamint az összes beolvasott karaktert! Az eredményt jelenítsük meg! (GYAK8_1) Használjuk az alábbi definíciót!

```
FILE *be, *ki;
int i, a_db = 0, e_db = 0, szokoz_db = 0, ossz_kar = 0;
char ki_szoveg[80], be_szoveg[80];
...
for (i = 0; be_szoveg[i]; i++)
{  switch (be_szoveg[i])
   {
     case 'a': a_db++; break;
     ...
```

A program futásának eredménye:

```
Szoveg: Szep ez a szalmakalap.
a karakterek szama: 5
e karakterek szama: 2
szokoz karakterek szama: 3
osszes karakter szama: 23
```

2. Olvassuk be a *SZERSZAM.TXT* állományt, amely raktáradatokat tartalmaz! Számláljuk meg, hogy hányfajta selejtes szerszám van a raktárban (*selejt_fajta_db*) és mennyi ezen szerszámoknak összes darabszáma (*selejt_db*), mennyi azon nem selejtes szerszámok darabszáma, melyeknek darabszámuk a raktárba 5-nél több (*nem_selejt_db*), valamint számláljuk meg a raktárban lévő összes (selejtes és nem selejtes) szerszám darabszámát is (*ossz_db*). Az eredményeket jelenítsük meg! (GYAK8_2)

Használjuk az alábbi definíciót!

```
typedef struct { char szerszam[10];
                 int selejt;  /* 1 selejt, 0 nem selejt */
                 int db;
               }raktar;
int main()
{ FILE *fr;
  raktar r;
  char sz[10];
  int selejt_db=0, selejt_fajta_db=0,nem_selejt_db=0,ossz_db=0;
  . . .
  fscanf(fr,"%10s %d %d\n",r.szerszam, &r.selejt, &r.db)
  . . .
}
```

A program futásának eredménye:

```
szerszam.txt allomany tartalma:
Szerszam   selejt darab
kalapacs       0    10
furesz         0     8
veso           1     2
fogo           1     2
furo           0     3

Selejtes szerszamfajtak szama: 2
Selejtes szerszamok darabszama: 4
Nem selejtes szerszamok darabszama (db>5): 18
Osszes szerszam darabszama: 25
```

3. Használjuk az alábbi definíciót!

```
typedef struct { char azonosito[12];
                 int db;
                 double ara;
               }aru;
int main()
{ FILE *faru;
  aru rk;
  double aruk_erteke = 0;
  int aruk_osszdb = 0;
  . . .
}
```

Olvassuk be az *ARUK.TXT* állomány tartalmát, adjuk össze az áruk darabszámát és számítsuk ki a raktárban lévő áruk értékét (áru darabszáma szorozva az áru értékével az összes árura nézve)! (GYAK8_3)

A program futásának eredménye:

```
aruk.txt allomany tartalma:
Aru azonosito darab     ara
asztal           2  1500.00
szek             3   500.00
szekreny         1  5000.00
fotel            3   700.00

Aruk ossz erteke: 11600.00
Aruk ossz darabszama: 9
```

9. Gyakorlat

Rövid elméleti összefoglaló

- **Bináris állományok**

A bináris állományokat bájtonként vagy blokkonként egyaránt kezelhetjük.

Az állomány megnyitása

A szöveges állományokhoz hasonlóan a háttértáron lévő fájl tartalmához csak a megnyitás után férhetünk hozzá. A megnyitás hozza létre a kapcsolatot a fájlmutató és az operációs rendszer állománya között. A megnyitáskor definiálódik az állomány típusa és a hozzáférés módja:

```
FILE *fopen(const char *fajlnev, const char *mod);
```

A *fajlnev* paraméter helyén adjuk meg az állomány operációs rendszerben használt nevét, melyben a teljes elérési útvonal is szerepelhet.
A *mod* paraméter szintén sztring, amely a fájl elérését és típusát meghatározza.

Bináris	*A **mod** paraméter jelentése*
"rb"	Létező állomány megnyitása.
"wb"	Új állomány megnyitása. Ha az állomány már létezik, akkor a tartalma elvész.
"ar"	Az állomány megnyitása a végéhez való hozzáírásra, tehát az állomány bővítése. Ha a fájl nem létezik, akkor megnyitáskor létrejön.
"r+b"	A létező állomány megnyitása írásra és olvasásra (*update*).
"w+r"	Új fájl megnyitása írásra és olvasásra (*update*). Ha az állomány már létezik, akkor a tartalma elvész.
"a+r"	Az állomány megnyitása a végén végzett írásra és olvasásra (*update*). Ha a fájl nem létezik, akkor most létrejön.

Ha az olvasni kívánt állomány nem létezik, akkor hibajelzést adunk, például:

```
if ((f = fopen("Utazas.dat","rb")) == NULL)
{
   fprintf(stderr,"Utazas.dat allomany nem letezik!\n");
   . . .
}
```

Bináris módú adatfolyam esetén nincs '\n' konverzió. Tehát a '\n' karakter kiírásakor mindig 13 kódú karakter íródik ki. Az operációs rendszerben a '\n' megfelelő kódsorozat is konvertálás nélkül kerül olvasásra. Tehát például a Windows rendszerekben bináris adatfolyamban érkező CR LF sorozat (13 10) a '\n'-re történő konverzió (1 karakter) helyett változatlan formában kerül beolvasásra.

Memóriablokk- írása és olvasása

A memóriablokkok állományba való írása és állományból történő visszaolvasása az *fwrite()*, illetve az *fread()* függvények használhatók, melyek prototípusa:

```
size_t fwrite(const void *ptr, size_t meret, size_t n,
              FILE* adatfolyam);
```

```
size_t fread(const void *ptr, size_t meret, size_t n,
             FILE* adatfolyam);
```

Mindkét függvény adott *meret* méretű adatelemekkel dolgozik, amelyekből *n* darabot ír vagy olvas a hívás során. Az adatokat tartalmazó memóriaterület kezdetét a *ptr* mutató jelöli ki.
A függvények visszatérési értéke a kiírt, illetve a beolvasott elemek számát (nem bájtok) adja meg. Ha ez az érték kisebb, mint az általunk kijelölt (*n*), akkor fájlvége, vagy fájlhiba eseményre lehet következtetni.

Pozicionálás az állományban
Az állomány elejére a ***rewind()*** függvénnyel pozicionálunk:

```
rewind(f);
```

Tetszőleges pozíciót választhatunk ki az ***fseek()*** függvénnyel:

```
int fseek(FILE *adatfolyam, long tavolsag, int honnan);
```

A *tavolsag* az állománypozíció relatív távolságát tartalmazza, a *honnan* paraméter által kijelölt pozícióhoz képest. A *honnan* paraméter lehetséges értékei:

SEEK_CUR	–	az aktuális pozícióhoz képest,
SEEK_END	–	a fájl végéhez képest,
SEEK_SET	–	a fájl elejéhez képest.

Az aktuális pozíciót lekérdezhetjük az ***ftell()*** függvénnyel:

```
long ftell(FILE *adatfolyam);
```

A MINTA9_01 mintafeladat az *Ir()* függvény a ponttal lezárt szöveget karakterenként a megadott (SZOVEG.DAT) állományba írja. Az *Olvas()* függvény az állományt újra megnyitja, a tartalmát karakterenként olvassa, és megszámlálja a magánhangzók, valamint az egyéb karakterek számát.

```
/*MINTA9_01*/
#include <stdio.h>
void Ir(char Fnev[]);
void Olvas(char Fnev[], int *egyeb, int *mgh);

int main()
{
  char Fnev[20], c;
  int egyebkar, maganh;
  printf("Allomany neve: "); scanf("%s",Fnev);
  Ir(Fnev);
  Olvas(Fnev, &egyebkar, &maganh);
  printf("\n\nMaganhangzok szama: %d\n",maganh);
  printf("Egyeb karakterek  : %d\n",egyebkar);
  fflush(stdin); getchar();
  return 0;
}
```

```
void Ir(char Fnev[])
{
  FILE *f;
  char ch;
  f = fopen(Fnev,"wb");
  printf("\nSzoveg (. a vege): \n");
  ch = '*';
  while (ch != '.')
  {
    ch = getc(stdin);
    fwrite(&ch, sizeof(ch), 1, f);
  }
  fclose(f);
}
void Olvas(char Fnev[], int *egyeb, int *mgh)
{
  FILE *f;
  char ch;
  if ((f = fopen(Fnev, "rb")) == NULL)
  {
    fprintf(stderr,"A %s allomany nem letezik!\n");
    getchar();
  }
  else
  {
    *egyeb = 0; *mgh = 0;
    printf("\nAllomanybol olvasva:");
    while (fread(&ch, sizeof(ch),1,f) == 1)
    {
      printf("%c",ch);
      if (ch == 'a' || ch == 'A' || ch == 'e' ||
          ch == 'E' || ch == 'i' || ch == 'I' ||
          ch == 'o' || ch == 'O' || ch == 'u' ||
          ch == 'U') (*mgh)++;
      else (*egyeb)++;
    }
    fclose(f);
  }
}
```

A program futásának eredménye:	
Allomany neve: szoveg.dat	Allomanybol olvasva: Ma jo lesz az ido.
Szoveg (. a vege): Ma jo lesz az ido.	Maganhangzok szama: 6 Egyeb karakterek : 13

String olvasása és írása

```
char *fgets(char *s, int n, FILE *adatfolyam);
int fputs(const char *s, FILE *str adatfolyam eam);
```

A MINTA9_02 mintafeladat függvényei:

AadatIr() a megadott darabszámú egész típusú adatokat a megadott *(ADATOK.DAT)* állományba írja.

AdatOlvas() az *ADATOK.DAT* állományból beolvassa az adatokat, és tömbben tárolja. Megszámlálja a pozitív, negatív és zérus számokat, valamint visszaadja az adatelemek számát.

OsszegSzamol() az adatok összegét visszatérési értékként, az átlagát pedig a paraméterlistán adja vissza.

```
/*MINTA9_02*/
#include <stdio.h>

typedef int tomb[100];
void AdatIr(char Fnev[]);
void AdatOlvas(char Fnev[],tomb t,int *poz,
               int *neg,int *zero,int *db);
int OsszegSzamol(tomb t, int n, double *atl);
int main()
{
  tomb x;
  int x_db , pozitiv, negativ, zerus, osszeg;
  double atlag;
  char Fnev[20];
  printf("Allomany neve: "); scanf("%s",Fnev);
  AdatIr(Fnev);
  AdatOlvas(Fnev,x,&pozitiv,&negativ,&zerus,&x_db);
  osszeg = OsszegSzamol(x,x_db,&atlag);
  printf("\nPozitiv adatok szama: %d\n",pozitiv);
  printf("Negativ adatok szama: %d\n",negativ);
  printf("Zerus adatok szama  : %d\n",zerus);
  printf("\nAdatok szama: %d\n",x_db);
  printf("Adatok osszege: %d\n",osszeg);
  printf("Adatok atlaga : %lf\n",atlag);
  fflush(stdin); getchar();
  return 0;
}
void AdatIr(char Fnev[])
{
  FILE *f;
  int i, k, n;
  f = fopen(Fnev, "wb");
  printf("Adatok szama: ");
  scanf("%d",&n);
  for (i = 0; i<n; i++)
  {
    printf("%d. adat: ",i); scanf("%d",&k);
    fwrite(&k, sizeof(int),1,f);
  }
  fclose(f);
}
```

```
void AdatOlvas(char Fnev[],tomb t,int *poz,
               int *neg,int *zero,int *db)
{ FILE *f;
  int i, k;
  if ((f = fopen(Fnev,"rb")) == NULL)
  {
    fprintf(stderr,"%s allomany nem letezik!\n",Fnev);
    getchar();
  }
  *db = 0; *poz = 0; *neg = 0; *zero = 0;

  while (fread(&k,sizeof(int),1,f) == 1)
  { (*db)++; i = *db;
    t[i] = k;
    if (t[i] < 0) (*neg)++;
    if (t[i] == 0) (*zero)++;
    if (t[i] > 0) (*poz)++;
  }
  fclose(f);
}
int OsszegSzamol(tomb t, int n, double *atl)
{ int i, ossz = 0;
  for (i = 0; i<n; i++)
  {
    ossz += t[i];
  }
  *atl = (double)ossz/n;
  return ossz;
}
```

A program futásának eredménye:

```
Allomany neve: adatok.dat
Adatok szama: 5
0. adat: 3
1. adat: -2
2. adat: 4
3. adat: -1
4. adat: 0
```

```
Pozitiv adatok szama: 2
Negativ adatok szama: 2
Zerus adatok szama  : 1

Adatok szama: 5
Adatok osszege: 4
Adatok atlaga : 0.800000
```

- **Struktúra adatainak kiírása bináris állományba**

A *Doboz* struktúra adatait *DOBOZOK.DAT* nevű bináris állományba írjuk.

```
/*MINTA9_03*/
#include <stdio.h>

typedef struct d {
                   double a,b,c; /*a doboz oldalai*/
                   double terf;
                   struct d *elozo;
                   struct d *kov;
                 } Doboz;
```

```
int main()
{
  int i,n;
  Doboz uj;
  FILE *fd;
  fd = fopen("dobozok.dat","wb");
  printf("Dobozok szama: ");
  scanf("%d",&n);
  for (i = 0; i<n; i++)
  {
    printf("\nA oldal: "); scanf("%lf",&uj.a);
    printf("B oldal: "); scanf("%lf",&uj.b);
    printf("C oldal: "); scanf("%lf",&uj.c);
    uj.terf = uj.a * uj.b * uj.c;
    uj.kov = NULL;
    uj.elozo = NULL;
    fwrite(&uj,sizeof(Doboz),1,fd);
  }
  fclose(fd);
  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
Dobozok szama: 3

A oldal: 1
B oldal: 2
C oldal: 3

A oldal: 4
B oldal: 5
C oldal: 6

A oldal: 2
B oldal: 2
C oldal: 2
```

- **Struktúra adatainak visszaolvasása bináris állományból**

A *DobozFOlvas()* függvényben *DOBOZOK.DAT* nevű bináris fájlból beolvassuk a *Doboz* struktúra adatait, és a memóriában láncoljuk. Listázhatjuk az adatokat, megkereshetjük a legnagyobb térfogatú dobozt. Ezek a függvények korábbi feladatban már szerepeltek.

```
/*MINTA9_04*/
#include <stdio.h>
#include <process.h>
typedef struct d {
                   double a,b,c;  /* a doboz oldalai */
                   double terf;
                   struct d *elozo;
                   struct d *kov;
                  } Doboz;
Doboz *DobozFOlvas(Doboz **p);
void ListaElore(Doboz *p);
void KeresMaxDoboz(Doboz *p);
...

Doboz *DobozFOlvas(Doboz **p)
{
  Doboz z, *uj, *elso = NULL, *utolso = NULL;
  FILE *fd;
  if ((fd = fopen("dobozok.dat","rb")) == NULL)
  {
    fprintf(stderr,"A dobozok.dat allomany nem letezik!\n");
    getchar();
  }
```

```
   else
  {
   while (fread(&z,sizeof(Doboz),1,fd) == 1)
   {  uj = (Doboz*)malloc(sizeof(Doboz));
      if (uj == NULL)
      { printf("Memoriafoglalas nem sikerult!\n");getchar();exit(1); }
      uj->a = z.a;
      uj->b = z.b;
      uj->c = z.c;
      uj->terf = z.terf;
      uj->kov = NULL;
      uj->elozo = NULL;
      if (elso == NULL) elso = uj;
      else
      {
        utolso->kov = uj;
        uj->elozo = utolso;
      }
      utolso = uj;
    }
    fclose(fd);
  }
  *p = utolso;
  printf("dobozok.dat beolvasasa befejezodott\n");
  return elso;
}
```

A program futásának eredménye:

```
dobozok.dat beolvasasa befejezodott

Listazas elore
 A oldal   B oldal   C oldal  Terfogat
    1.00      2.00      3.00      6.00
    4.00      5.00      6.00    120.00
    2.00      2.00      2.00      8.00

 A oldal   B oldal   C oldal  Max terfogat
    4.00      5.00      6.00    120.00
```

A *MINTA9_05* mintafeladat bemutatja a ***rewind()*** *és* ***fseek()*** függvények használatát, valamint ***ftell()*** hívásával a pozíció leolvasását.

```
/*MINTA9_05*/
#include <stdio.h>
#include <stdlib.h>
typedef struct d {
                  double a,b,c;  /* a doboz oldalai */
                  double terf;
                  struct d *elozo;
                  struct d *kov;
                 } Doboz;
 int main()
{ int i,n;
  long meret, pozicio;
```

```
  Doboz uj;
  FILE *fd;
  fd = fopen("dobozok.dat","wb"); /* megnyitja írásra az állományt */
  printf("Dobozok szama: ");  /* beolvassa a dobozok számát */
  scanf("%d",&n);
  for (i = 0; i<n; i++)
  { printf("%d. blokk",i);  /* beolvassa a dobozok adatait */
    printf("\nA oldal: "); scanf("%lf",&uj.a);
    printf("B oldal: "); scanf("%lf",&uj.b);
    printf("C oldal: "); scanf("%lf",&uj.c);
    uj.terf = uj.a * uj.b * uj.c;
    uj.kov = NULL;
    uj.elozo = NULL;
    fwrite(&uj,sizeof(Doboz),1,fd); /* kiírja blokkokat */
  }
  fclose(fd);   /* lezárja az állományt */
  printf("\n0. blokk olvasasa");
  fd = fopen("dobozok.dat","rb"); /* megnyitja olvasásra */
  fread(&uj,sizeof(Doboz),1,fd);  /* beolvassa az első rekordot */
  printf("\nA oldal: %lf\n",uj.a); /* megjeleníti az adatokat */
  printf("B oldal: %lf\n",uj.b);
  printf("C oldal: %lf\n",uj.c);
  rewind(fd);  /* visszaáll az első rekord elé */
  fseek(fd,2*sizeof(Doboz),SEEK_SET); /* 1. rekord elé áll */
  pozicio = ftell(fd)/sizeof(Doboz);  /* pozicio leolvasása */
  printf("\n%ld. blokkra pozicionalas\n",pozicio);
  fread(&uj,sizeof(Doboz),1,fd); /*beolvassa a 1. sorszámú rekordot */
  printf("\nA oldal: %lf\n",uj.a); /* megjeleníti az adatait */
  printf("B oldal: %lf\n",uj.b);
  printf("C oldal: %lf\n",uj.c);
  fclose(fd); /*lezárja az állományt */
  fflush(stdin);getchar();
  return 0;
}
```

A rekordok (azaz a blokkok) 0 sorszámmal kezdődnek.

A program futásának eredménye:

```
Dobozok szama: 3
0. blokk
A oldal: 1
B oldal: 2
C oldal: 3
1. blokk
A oldal: 4
B oldal: 5
C oldal: 6
2. blokk
A oldal: 7
B oldal: 8
C oldal: 9
0. blokk olvasasa
A oldal: 1.000000
B oldal: 2.000000
C oldal: 3.000000

2. blokkra pozicionalas

A oldal: 7.000000
B oldal: 8.000000
C oldal: 9.000000
```

A *MINTA9_06* mintafeladat az *Utaztato* struktúra adatait írja ki az *UTAZAS.DAT* bináris állományba.

```
/*MINTA9_06*/
#include <stdio.h>
typedef struct {
                  char uticel[12];
                  int napok_szama;
                  int utasok_szama;
                } Utaztato;
int main()
{ Utaztato u;
  FILE *f;
  int i, n;
  f = fopen("Utazas.dat","wb");
  printf("Utazasi programok szama: ");
  scanf("%d",&n);
  for (i = 0; i<n; i++)
  { printf("Uticel : "); scanf("%s",u.uticel);
    printf("Napok szama  : "); scanf("%d",&u.napok_szama);
    printf("Utasok szama : "); scanf("%d",&u.utasok_szama);
    fwrite(&u,sizeof(Utaztato),1,f);
  }
  fclose(f);
  fflush(stdin); getchar();
  return 0;
}
```

A *MINTA9_07* mintafeladat beolvassa az *UTAZAS.DAT* bináris állományban tárolt *Utaztato* típusú rekordokat, megszámlálja azoknak az utasoknak a számát, akik Berlinbe utaznak, valamint azoknak az utasoknak a számát, akik 7 napot töltenek el, valamint az összes utazók számát.

```
/*MINTA9_07*/
#include <stdio.h>

#include <string.h>
typedef struct {
                  char uticel[12];
                  int napok_szama;
                  int utasok_szama;
                } Utaztato;
int main()
{ Utaztato u;
  FILE *f;
  int i, uticel_db = 0, hetnapos_db = 0, ossz_utas = 0;
  if ((f = fopen("Utazas.dat","rb")) == NULL)
  { fprintf(stderr,"Utazas.dat allomany nem letezik!\n");
    getchar();
  }
  else
```

```
  { while (fread(&u, sizeof(Utaztato),1,f) == 1)
    { printf("Uticel : %s\n",u.uticel);
      printf("Napok szama  : %d\n",u.napok_szama);
      printf("Utasok szama : %d\n\n",u.utasok_szama);
      if (strcmp(u.uticel, "Berlin") == 0)
        uticel_db += u.utasok_szama;
      if (u.napok_szama == 7)
        hetnapos_db += u.utasok_szama;
      ossz_utas += u.utasok_szama;
    }
    fclose(f);
    printf("Osszes utas: %d\n",ossz_utas);
    printf("Berlinbe utazok szama: %d\n",uticel_db);
    printf("7 napos ut utasainak szama: %d\n",hetnapos_db);
  }
  getchar();
  return 0;
}
```

A program futásának eredménye:

```
Uticel : Berlin
Napok szama  : 4
Utasok szama : 12

Uticel : Berlin
Napok szama  : 3
Utasok szama : 8

Uticel : Praga
Napok szama  : 7
Utasok szama : 12

Uticel : London
Napok szama  : 5
Utasok szama : 15

Uticel : Parizs
Napok szama  : 7
Utasok szama : 14

Osszes utas: 61
Berlinbe utazok szama: 20
7 napos ut utasainak szama: 26
```

- **Tömbadatok bináris állományba mentése és visszaolvasása onnan**

A bináris állományok felépítése megegyezik az egydimenziós tömbök struktúrájával a memóriában. Ezért e kétféle adattárolási megoldás közötti adatcsere nagyon gyorsan és kényelmesen elvégezhető. A *MINTA9_08* példában a billentyűzetről beolvasott egész számokat tömbben tároljuk, majd a teljes tömböt az ***fwrite()*** függvény egyetlen hívásával bináris állományba írjuk. A fájl 0. adatelemében az elemek számát tároljuk.

```
/*MINTA9_08*/
#include <stdio.h>
#include <stdlib.h>

#define MAXN 123

int main()
{
  int tomb[MAXN];
  int i,n;
  FILE *fd;
```

```
  do {
     printf("A tomb elemeinek szama [1..%d]: ",MAXN);
     scanf("%d",&n);
  } while (n>MAXN || n<1);
  for (i = 0; i<n; i++)
  {
    printf("Tomb[%3d] = ",i);
    scanf("%d",&tomb[i]);
  }

  /* A tömb fájlba írása - az elemek számát a tömb elemei követik */
  printf("\nA tomb elemeinek fajlba irasa:\n");
  fd = fopen("Tomb.dat","wb");
  fwrite(&n,sizeof(int),1,fd);     /* az elemek száma */
  fwrite(tomb,sizeof(int),n,fd);   /* a tömb elemei   */
  fclose(fd);

  /* A tömb visszaolvasása bináris fájlból */
  printf("\nA tomb elemeinek visszaolvasasa:\n");
  fd = fopen("Tomb.dat","rb");
  fread(&n,sizeof(int),1,fd);     /* az elemek száma */
  fread(tomb,sizeof(int),n,fd);   /* a tömb elemei   */
  fclose(fd);

  printf("\nA visszaolvasott tombelemek:\n");
  /* Az elemek megjelenítése */
  for (i=0; i<n; i++)
    printf("%5d", tomb[i]);

  fflush(stdin);getchar();
  return 0;
}
```

A program futásának eredménye:

```
A tomb elemeinek szama [1..123]: 4
Tomb[  0] = 12
Tomb[  1] = 56
Tomb[  2] = -2
Tomb[  3] = 78

A tomb elemeinek fajlba irasa:

A tomb elemeinek visszaolvasasa:

A visszaolvasott tombelemek:
   12   56   -2   78
```

Feladatok

1. Írjunk ki adott darabszámú *Verseny* struktúrát adatait a *VERSENY.DAT* bináris állományba. Olvassuk vissza a rekordokat, és számláljuk meg, hogy hány olyan *sportag* van, ahol a versenyzők száma nagyobb, mint 20 fő! Keressük meg a legtöbb díjat és a hozzá tartozó sportágat! Jelenítsük meg az eredményeket! (GYAK9_1)

Használjuk az alábbi definíciót:

```
typedef struct {
                  char sportag[10];
                  int versenyzokszama;
                  int dijak;
                } Verseny;
int main()
{ Verseny v;
  FILE *fv;
  int i, n, db_20 = 0, dij = 0;
  char sportag_dij[10];
  . . .
}
```

A program futásának eredménye:

```
Sportagak szama: 3

Sportag : kenu
Versenyzok szama  : 15
Dijak : 3

Sportag : kajak
Versenyzok szama  : 22
Dijak : 2

Sportag : vitorlas
Versenyzok szama  : 49
Dijak : 1

Versenyzok szama nagyobb, mint 20 : kajak
Versenyzok szama nagyobb, mint 20 : vitorlas

Legtobb dijat nyert sportag: kenu
Dijak szama: 3
```

2. Írjunk egy *MuveletIr()* függvényt, amely adott számú *Muvelet* típusú struktúra adattagjait kiírja a *MUVELET.DAT* állományba. A *Listaz()* függvény a *MUVELET.DAT* állományból beolvassa és megjeleníti a *Muvelet* típusú struktúrákat. (GYAK9_2)

```
typedef struct {
                  double adat1, adat2;
                  char muveletijel;
                  double ered;
                 }Muvelet;
void MuveletIr(void);
void Listaz(void);
```

A program futásának eredménye:

```
Muveletek szama: 4

1. adat: 2
2. adat: 3
Muveleti jel: +

1. adat: 4
2. adat: 5
```

```
Muveleti jel: -

1. adat: 6
2. adat: 7
Muveleti jel: *

1. adat: 8
2. adat: 4
Muveleti jel: /


Muvelet.dat allomany tartalma:
    2.00  +    3.00
    4.00  -    5.00
    6.00  *    7.00
    8.00  /    4.00
```

3. Használjuk az alábbi definíciót!

```
typedef struct {
                  double adat1, adat2;
                  char muveletijel;
                  double ered;
                 }Muvelet;
void MuveletOlvas(void);
```

Írjunk *MuveletOlvas()* függvényt, amely a *MUVELET.DAT* állományt beolvassa és végrehajtja az adatok által kijelölt műveleteket! (GYAK9_3)

A program futásának eredménye:

```
2.00  +    3.00 =     5.00
4.00  -    5.00 =    -1.00
6.00  *    7.00 =    42.00
8.00  /    4.00 =     2.00
```

10. Gyakorlat

Rövid elméleti összefoglaló

- **Többmodulos programok írása, projekt létrehozása**

Nagyobb méretű C programok készítése során a forrásprogramot több állományban (modulokban) helyezzük el, melyeket önállóan fordít a fordító. A modulokban definiált neveket kiajánlhatjuk más modulok számára az **extern** tárolási osztállyal, ami alapértelmezés a függvénynevek és a függvényeken kívül definiált változók esetén. Amennyiben az alapértelmezett **extern** nevek elérhetőségét a modulon belülre kívánjuk korlátozni, a **static** tárolási osztály kell használnunk. Mivel minden modul önállóan lefordítható kell legyen, a deklarációs (.h) állományok segítségével teremtjük meg a kapcsolatot a modulok között.

A *MINTA10_01* mintafeladat több modulból épül fel, ezért a fordításához projektet kell használnunk. A projekt előállításának lépései nagyban függenek a fejlesztői környezettől. Az általunk használt rendszerben az 1. Függelék ismerteti a szükséges lépéseket.

A példánkban a *MAIN.C* állomány tartalmazza a ***main()*** függvényt, és a *VEKTOR3D.C* fájlban találjuk a térbeli vektorok kezelését segítő definíciókat. Ahhoz, hogy más modulból is elérhetőek legyenek a modul külső definíciói, a szükséges deklarációkat és prototípusokat a *VEKTOR3D.H* fejállományban gyűjtöttük csokorba. Ezt a deklarációs fájlt mindkét C-modulba az ***#include*** előfordító utasítással építjük be, nem elfeledve, hogy saját állományról van szó (""):

```
#include "Vektor3d.h"
```

```
/* VEKTOR3D.H */
typedef struct {
  double x,y,z;
} Vektor3D;

/* Projektszintű változók deklarációja, ezt jelöli az extern */
extern const Vektor3D origo;
/* egységvektorok x-, y- és z-irányban */
extern const Vektor3D ei, ej, ek;

/* Projektszintű függvények prototípusa */
double Sszorzat(const Vektor3D a, const Vektor3D b);
Vektor3D VSzorzat(const Vektor3D a, const Vektor3D b);
void VKiir(const char *s, const Vektor3D v);
void Vegyseg(Vektor3D * p);

/* Modulszintű (statikus) függvény prototípusa */
static double VHossz(const Vektor3D a);
```

```
/* VEKTOR3D.C */

#include <stdio.h>
#include <math.h>
#include "Vektor3D.h"
```

```
/* Projektszintű változók létrehozása, nincs extern megadva */
const Vektor3D origo = {0, 0, 0}, ei = {1, 0, 0};
const Vektor3D    ej = {0, 1, 0}, ek = {0, 0, 1};

/* Modulszintű definíciók */
static const double pi = 3.14159265;

/* Két vektor skaláris szorzata */
double SSzorzat(const Vektor3D a, const Vektor3D b)
{
   return a.x*b.x + a.y*b.y + a.z*b.z;
}

/* Két vektor vektoriális szorzata */
Vektor3D VSzorzat(const Vektor3D a, const Vektor3D b)
{
   Vektor3D c;
   c.x = a.y*b.z - b.y*a.z;
   c.y = -(a.x*b.z - b.x*a.z);
   c.z = a.x*b.y - b.x*a.y;
   return c;
}

/* A vektor egységvektorrá való alakítása */
void VEgyseg(Vektor3D * p)
{
   double vh = Vhossz(*p);
   p->x = p->x / vh;
   p->y = p->y / vh;
   p->z = p->z / vh;
}

/* Vektor megjelenítése */
void VKiir(const char *s, const Vektor3D v)
{
   printf("(%s%5.2lf,%5.2lf,%5.2lf)\n", s, v.x, v.y, v.z);
}

/* Modulszintű függvény definíciója - a vektor hossza */
static double Vhossz(const Vektor3D a)
{
  return sqrt(a.x*a.x+a.y*a.y+a.z*a.z);
}
```

```
/*MINTA10_01*/
#include <stdio.h>
#include "Vektor3D.h"

int main()
{
  Vektor3D a = ek;
  Vektor3D b = {  1 , 2 , 3};
  Vektor3D c = { 12, 79, 23};
```

```
  VKiir("a      = ", a);
  VKiir("b      = ", b);
  VKiir("a x b = ", VSzorzat(a,b));
  printf(" a . b = %5.2lf\n", SSzorzat(a,b));

  VKiir("c      = ", c);
  VEgyseg(&c);
  VKiir("c      = ", c);

  getchar();
  return 0;
}
```

A program futásának eredménye:

```
(a      =  0.00, 0.00, 1.00)
(b      =  1.00, 2.00, 3.00)
(a x b = -2.00, 1.00, 0.00)
 a . b =  3.00
(c      = 12.00,79.00,23.00)
(c      =  0.14, 0.95, 0.28)
```

Feladat

1. Bővítsük a MINTA10_01 mintafeladat vektorműveleteit a vektorok összeadásával és a vektorok beolvasásával! (GYAK10_1)

2. Készítsen egyetlen, dinamikus helyfoglalású egész vektor használatát biztonságossá tevő programmodult! Mutassa be a vektor használatához szükséges függvényeket egy főmodul kidolgozásával! (GYAK10_2)

   Használja az alábbi deklarációkat!

   ```
   int Lefoglal(int m);
   void Felszabadit();
   int Elemszam();
   int * Elem(int index);
   void Feltolt(int elem);
   void Kiir();
   ```

3. Egészítse ki a GYAK10_2 projektet egy olyan modullal, melynek segítségével a vektor tartalma bináris állományba menthető, illetve onnan visszaolvasható! (GYAK10_3)

11. Gyakorlat

A C nyelv néhány speciális alkalmazása

- **A union típus használata**

A **union** típust gyakran felesleges örökségnek tekintik a C nyelv "őskorából", azonban e fejezetben két felhasználási módot is bemutatunk. Először megnézzük, hogy lehet **union** segítségével gyorsan bizonyos átalakításokat elvégezni. A *MINTA11_01* mintafeladatban egy hosszú egész értéket darabolunk ketté, illetve 4 bájttá, valamint összerakunk bájtokból.

```
/*MINTA11_01*/
#include <stdio.h>

typedef union {
                unsigned char b[4];
                unsigned short int w[2];
                 unsigned long int d;
} konverter;

int main()
{
   konverter k;
   int i;
   k.d = 0x012F35F7;
   printf("A long szam reszekre bontasa: \n");
   printf("\nA long szam: %08X\n", k.d);

   printf("\nA szam ket szava: ");
   for (i=0; i<2; i++)
     printf("\t%04X", k.w[i]);
   putchar('\n');

   printf("\nA szam negy bajtja: ");
   for (i=0; i<4; i++)
     printf("\t%02X", k.b[i]);
   putchar('\n');

   printf("\n\nA long szam felepitese bajtokbol: \n");
   printf("\nA szam negy bajtja: ");
   for (i=0; i<4; i++) {
      k.b[i] = 0xB<<i;
      printf("\t%02X", k.b[i]);
   }
   putchar('\n');
   printf("\nA long szam: %08X\n", k.d);

   getchar();
   return 0;
}
```

A program futásának eredménye:

```
A long szam reszekre bontasa:
A long szam: 012F35F7
A szam ket szava:        35F7     012F
A szam negy bajtja:      F7       35       2F       01
A long szam felepitese bajtokbol:
A szam negy bajtja:      0B       16       2C       58
A long szam: 582C160B
```

Gyakran előfordul, hogy közel azonos adatokat kell egyetlen adatstruktúrában tárolnunk. Néhány nyelv a variáns rekordokkal támogatja az ilyen feladatok megoldását. C nyelvben a struktúrába ágyazott **union** típus használatával érjük el célunkat. A *MINTA11_02* mintafeladatban kör- és háromszögadatokat tárolunk közös struktúrában egy típusjelző adattag (*alakzat*) kíséretében.

```
/*MINTA11_02*/
#include <stdio.h>
#include <stdlib.h>

#include <math.h>
enum Tipus {kor, haromszog};

struct Alakzat {
   enum Tipus alakzat;
   union {
     double r;
     struct {
         double a,b,c;
     } oldalak;
   } u;
};

void Beolvas(struct Alakzat * a, enum Tipus t);
double Terulet(struct Alakzat a);

int main()
{
  struct Alakzat k, h;
  Beolvas(&k, kor);
  printf("A kor terulete: %8.4lf\n", Terulet(k));
  Beolvas(&h, haromszog);
  printf("A haromszog terulete: %8.4lf\n", Terulet(h));
  getchar();
  return 0;
}
```

```
/* Valós szám biztonságos beolvasása */
double GetDouble() {
   char s[81];
   double d;
   gets(s);
   if(!sscanf(s,"%lf",&d)) d=0;
   return d;
}

void Beolvas(struct Alakzat * a, enum Tipus t) {
    a->alakzat=t;
    switch (t) {
       case kor:
          printf("\nA kor sugara: ");
          a->u.r = GetDouble();
          break;
       case haromszog:
          printf("\nA haromszog a oldala: ");
          a->u.oldalak.a = GetDouble();
          printf("A haromszog b oldala: ");
          a->u.oldalak.b = GetDouble();
          printf("A haromszog c oldala: ");
          a->u.oldalak.c = GetDouble();
          break;
    }
}

double Terulet(struct Alakzat a)
{
    double t=0, s, p;
    switch (a.alakzat)
    {
       case kor:
          t = a.u.r*a.u.r*(4*atan(1));
          break;
       case haromszog:
          s=(a.u.oldalak.a+a.u.oldalak.b+a.u.oldalak.c)/2;
          p=s*(s-a.u.oldalak.a)*(s-a.u.oldalak.b)*(s-a.u.oldalak.c);
          /* a háromszögegyenlőtlenség ellenőrzése */
          t = p>0 ? sqrt(p) : 0;
          break;
    }
    return t;
 }
```

A program futásának eredménye:

```
A kor sugara: 1
A kor terulete:   3.1416

A haromszog a oldala: 3
A haromszog b oldala: 4
A haromszog c oldala: 5
A haromszog terulete:   6.0000
```

- **Rekurzív megoldások programozása**

A rekurzió, vagyis az önmagukat hívó függvények alkalmazása, érdekes és látványos programozási megoldásokra ad lehetőséget. Az esetek többségében azonban az így elkészített függvények nem elég hatékonyak, lassúak és sok memóriát igényelnek. Belátható, hogy a rekurzív algoritmusok mindig átfogalmazhatók ciklusokra épülő iteratív megoldásokká. Nézzünk először egy példát, amikor a rekurzív és az iteratív megoldás között nincs lényegi különbség. A *MINTA11_03* programban a bináris keresést valósítjuk meg mindkét módon.

```
/*MINTA11_03*/
#include <stdio.h>

/* Bináris keresés iteráció alkalmazásával */
int binkeri(int ertek, int tomb[], int n);

/* Bináris keresés rekurzió alkalmazásával */
int binkerr(int ertek, int tomb[], int n);

int main()
{
  int a[12] = {1, 4, 7, 8, 9, 12, 17, 23, 25, 27, 31, 35};
  int i;

  printf("\nIndex          :");
  for (i=0; i<12; i++)
    printf("%4d", i);
  printf("\nRendezett tomb :");
  for (i=0; i<12; i++)
    printf("%4d", a[i]);
  printf("\n");

  printf("\nIteracioval:\n");
  printf("%5d elem indexe: %5d\n",  5, binkeri( 5, a, 12));
  printf("%5d elem indexe: %5d\n", 12, binkeri(12, a, 12));
  printf("%5d elem indexe: %5d\n", 23, binkeri(23, a, 12));
  printf("%5d elem indexe: %5d\n",  1, binkeri( 1, a, 12));
  printf("%5d elem indexe: %5d\n", 35, binkeri(35, a, 12));

  printf("\nRekurzioval:\n");
  printf("%5d elem indexe: %5d\n",  5, binkerr( 5, a, 12));
  printf("%5d elem indexe: %5d\n", 12, binkerr(12, a, 12));
  printf("%5d elem indexe: %5d\n", 23, binkerr(23, a, 12));
  printf("%5d elem indexe: %5d\n",  1, binkerr( 1, a, 12));
  printf("%5d elem indexe: %5d\n", 35, binkerr(35, a, 12));
  getchar();
  return 0;
}

/* Bináris keresés iteráció alkalmazásával */
int binkeri(int ertek, int tomb[], int n)
{
    int kozepso, also = 0, felso = n;
    while (also < felso)
    {
              kozepso = (also + felso) >> 1;
              if (ertek == tomb[kozepso])
                 return kozepso;
```

```
            if (ertek > tomb[kozepso])
                  /* ertek > tomb[kozepso] : jobbra megyünk */
                    also = kozepso + 1;
            else
                 /* különben balra megyünk */
                    felso = kozepso;
        }
        return -1;  /* nem talált */
}

/* Bináris keresés rekurzió alkalmazásával */
int bk(int ertek, int tomb[], int also, int felso)
{
   int kozepso = (also + felso) >> 1;
   if (ertek == tomb[kozepso])
       return kozepso;
   else
      if (also == felso)
          return -1;  /* nem talált */
      else
          if (ertek > tomb[kozepso])
                /* ertek > tomb[kozepso] : jobbra megyünk */
                 return bk(ertek, tomb, kozepso+1, felso);
          else
                /* különben balra megyünk */
              return bk(ertek, tomb, also, kozepso);
}

int binkerr(int ertek, int tomb[], int n)
{
    return bk(ertek, tomb, 0, n);
}
```

A program futásának eredménye:

```
Index          :   0   1   2   3   4   5   6   7   8   9  10  11
Rendezett tomb :   1   4   7   8   9  12  17  23  25  27  31  35

Iteracioval:
    5 elem indexe:    -1
   12 elem indexe:     5
   23 elem indexe:     7
    1 elem indexe:     0
   35 elem indexe:    11

Rekurzioval:
    5 elem indexe:    -1
   12 elem indexe:     5
   23 elem indexe:     7
    1 elem indexe:     0
   35 elem indexe:    11
```

A *MINTA11_04* mintapéldában determináns kiszámításához használjuk a rekurziót. A megoldás lényege, hogy az N-ed rendű determináns meghatározását N darab (N-1)-ed rendű determináns számítására vezetjük vissza. Tesszük ezt mindaddig, míg el nem jutunk a másodrendű

determinánsokig. A megoldás szép és áttekinthető, azonban nem elég hatékony. A számítás során a hívások fastruktúrában láncolódnak, ami sok futásidőt és memóriát igényel.

Például, a 4x4-es determináns számításakor 17-szer (1 darab 4x4-es, 4 darab 3x3-as és 4·3 darab 2x2-es mátrixszal) hívódik meg a *Det()* függvény, a hívások száma azonban az 5-öd rendű esetben 5·17+1 = 86-ra emelkedik. (Megjegyezzük, hogy 12x12-es mátrix esetén a *Det()* függvény, több mint 300 milliószor hívódik meg, ami perceket is igénybe vesz.)

```
/*MINTA11_04*/
#include <stdio.h>
#include <stdlib.h>

typedef double Matrix[12][12];

double Det(Matrix m, int n);

void MatrixKiir(Matrix m, int n);

int main()
{
  Matrix m2 = {{1, 2},
               {2, 3}};

  Matrix m3 = {{1, 2, 3},
               {2, 1, 1},
               {1, 0, 1}};

  Matrix m4 = {{ 2, 0, 4, 3},
               {-1, 2, 6, 1},
               {10, 3, 4,-2},
               { 2, 1, 4, 0}};

  MatrixKiir(m2,2);
  printf("Det(m2) = %lf\n\n",Det(m2,2));

  MatrixKiir(m3,3);
  printf("Det(m3) = %lf\n\n",Det(m3,3));

  MatrixKiir(m4,4);
  printf("Det(m4) = %lf\n\n",Det(m4,4));

  getchar();
  return 0;
}

void MatrixKiir(Matrix m, int n)
{
   int i,j;
   for(i=0; i<n; i++) {
       for (j=0; j<n; j++)
         printf("%6.2lf",m[i][j]);
       putch('\n');
   }
}
```

```
double Det(Matrix m, int n)
{
  int i, j, k, q;
  double s;
  Matrix x;

  if (n==2)
    return m[0][0]*m[1][1]-m[1][0]*m[0][1];

  s = 0;
  for (k=0; k<n; k++)  /* n darab aldeterminánsok */
  {
        /* az almátrixok létrehozása */
        for (i=1; i<n; i++)     /* sorok */
        {
           q = -1;
           for (j=0; j<n; j++) /* oszlopok */
            if (j!=k)
              x[i-1][++q] = m[i][j];
        }
        s+=(k % 2 ? -1 : 1) * m[0][k] * Det(x,n-1);
  }
  return s;
}
```

A program futásának eredménye:

```
  1.00  2.00
  2.00  3.00
Det(m2) = -1.000000

  1.00  2.00  3.00
  2.00  1.00  1.00
  1.00  0.00  1.00
Det(m3) = -4.000000

  2.00  0.00  4.00  3.00
 -1.00  2.00  6.00  1.00
 10.00  3.00  4.00 -2.00
  2.00  1.00  4.00  0.00
Det(m4) = 144.000000
```

- **Halmazok kezelése**

Mint tudjuk, a halmaztípus hiányzik a C nyelv eszköztárából, azonban ez nem jelenti azt, hogy le kell mondanunk a használatáról. A *MINTA11_05* mintapéldában karakterhalmazok kezelését valósítjuk meg. A megoldás lényege, az elem-e információ tárolására egyetlen bitet használunk. Ennek következtében minden *Halmaz* 32 bájtot foglal el a memóriából.

```
typedef struct HALMAZ {
  unsigned char p[256/8];  /* a tömb mérete 32 byte */
} Halmaz;
```

A megoldásban nyomon követhetjük a legfontosabb halmazműveleteket (egyesítés, metszet, különbség, eleme- és részhalmaz-vizsgálat), és példát láthatunk a többmodulok projekt kialakítására.

```
/*HALMAZ.H*/
#define bytehossz(n) ((n) % 8 ? 1 : 0) + (n) / 8
#define max(a,b) ((a)>(b) ? (a) : (b))
#define min(a,b) ((a)<(b) ? (a) : (b))

typedef struct HALMAZ {
  unsigned char p[256/8];  /* a tömb mérete 32 byte */
} Halmaz;

Halmaz UresHalmaz();
void AdElem(Halmaz * h, unsigned char e);
void AdElemek(Halmaz * h, unsigned char *s);
void TorolElem(Halmaz * h, unsigned char e);
int Eleme(Halmaz h, unsigned char e);
Halmaz Unio(Halmaz a, Halmaz b);
Halmaz Metszet(Halmaz a, Halmaz b);
Halmaz Kulonbseg(Halmaz a, Halmaz b);
void KiirHalmaz(char * s, Halmaz h);
int Szamossag(Halmaz h);
int ReszHalmaza(Halmaz a, Halmaz b);
```

```
/*HALMAZ.C*/
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include "halmaz.h"

Halmaz UresHalmaz() {
   Halmaz h;
   int i;
   for (i=0; i<32; i++)
     h.p[i]=0;
   return h;
}

void AdElem(Halmaz * h, unsigned char e) {
  int i;
  h->p[e / 8] |= 128>>e % 8;
}

void AdElemek(Halmaz * h, unsigned char *s)
{
  while(*s)
     AdElem(h, *s++);
}

void TorolElem(Halmaz *h, unsigned char e)
{
  h->p[e / 8] &= ~(128>>e % 8);
}
```

```
int Eleme(Halmaz h, unsigned char e)
{
  return h.p[e /8] & (128>>e % 8);
}

Halmaz Unio(Halmaz a, Halmaz b) {
  int i;
  Halmaz c;
  for (i=0; i<32; i++)
    c.p[i] = a.p[i] | b.p[i];
  return c;
}

Halmaz Metszet(Halmaz a, Halmaz b) {
  int i;
  Halmaz c;
  for (i=0; i<32; i++)
    c.p[i] = a.p[i] & b.p[i];
  return c;
}

Halmaz Kulonbseg(Halmaz a, Halmaz b) {
  Halmaz c;
  int i;
  for (i=0; i<32; i++)
    c.p[i] = a.p[i] & ~b.p[i];
  return c;
}

void KiirHalmaz(char * s, Halmaz h) {
  int i;
  printf("%s [",s);
  for (i=0; i<256; i++)
    if (Eleme(h,i))
       printf("%2c", i);
  printf(" ]\n");
}

int Szamossag(Halmaz h)
{
  int i, s=0;
  for (i=0; i<256; i++)
    if (Eleme(h,i))
       s++;
  return s;
}

int Reszhalmaza(Halmaz a, Halmaz b)
{
   Halmaz k=Kulonbseg(a,b);
   if (Szamossag(k))
     return 0;
   else
     return 1;
}
```

```
/*MINTA11_05*/
#include <stdio.h>
#include "halmaz.h"

int main()
{
  Halmaz a=UresHalmaz();
  Halmaz b=UresHalmaz();
  Halmaz c,d,e,s;

  AdElem(&a,'N');
  AdElemek(&a,"NATB");
  KiirHalmaz("A = ", a);
  printf("Az A halmaz elemeinek szama: %d\n", Szamossag(a));
  AdElem(&b,'A');
  AdElemek(&b,"LANCDI");
  TorolElem(&b,2);
  KiirHalmaz("\nB = ",b);
  printf("A B halmaz elemeinek szama: %d\n", Szamossag(b));
  c=Unio(a,b);
  KiirHalmaz("\nA U B = ",c);

  d=Metszet(a,b);
  KiirHalmaz("\nA x B = ",d);

  e=Kulonbseg(a,b);
  KiirHalmaz("\nA \\ B = ",e);
  if (Eleme(b,'N'))
    printf("\nN eleme a B halmaznak\n");
  else
    printf("\nN nem eleme a B halmaznak\n");

  s=UresHalmaz();
  KiirHalmaz("\nS = ",s);
  if (Reszhalmaza(s,b))
    printf("S reszhalmaza B-nek\n");
  else
    printf("S nem reszhalmaza B-nek\n");

  AdElemek(&s,"NA");
  KiirHalmaz("\nS = ",s);
  if (Reszhalmaza(s,b))
    printf("S reszhalmaza B-nek\n");
  else
    printf("S nem reszhalmaza B-nek\n");

  getchar();
  return 0;
}
```

A program futásának eredménye:

```
A =  [ A B N T ]
Az A halmaz elemeinek szama: 4

B =  [ A C D I L N ]
A B halmaz elemeinek szama: 6
```

```
A U B =  [ A B C D I L N T ]

A x B =  [ A N ]

A \ B =  [ B T ]

N eleme a B halmaznak

S =  [ ]
S reszhalmaza B-nek

S =  [ A N ]
S reszhalmaza B-nek
```

- **Rendezett sztringlista**

Végezetül a térjünk vissza a listakészítés témaköréhez! A *MINTA11_06* mintafeladatban legfeljebb 122 karakter hosszú karaktersorozatokat tárolunk a listaelemekben. A feltöltés során a kapcsolatok kialakításával biztosítjuk a beírt szövegsorok rendezettségét.

```
/*MINTA11_06*/
#include <stdio.h>
#include <assert.h>
#include <string.h>
#include <stdlib.h>

/* a listaelem típusa */
typedef struct ListaElem{
   char nev[123];
   struct ListaElem * ptr;
} telem ;

typedef telem * pelem;

/* új elem beszúrása a listába */
void ListaUjELem(pelem * bptr, const char nev0[]);

/* a lista végigjárása az elemek kiíratásával */
void ListaKiirasa(pelem bptr);

/* a lista felszabadítása */
void ListaTorlese(pelem bptr);

int main()
{
  pelem begptr, sptr, tptr;

  /* a sztringlista felépítése, a begptr a lista elejére mutat */
  begptr = NULL;

  ListaUjELem(&begptr, "Natalia");
  ListaUjELem(&begptr, "Lena");
  ListaUjELem(&begptr, "Julia");
  ListaUjELem(&begptr, "Anna");
  ListaUjELem(&begptr, "Adrienn");
```

```
  ListaKiirasa(begptr);
  ListaTorlese(begptr);
  begptr = NULL;

  getchar();
  return 0;
}

/* ------------ új elem beszúrása a listába  ------------ */
void ListaUjELem(pelem * bptr, const char nev0[])
{
    pelem sptr, tptr, newptr;
    /* helyfoglalás az új elem számára */
    newptr = (pelem)malloc(sizeof(telem));
    assert(newptr);
    strcpy(newptr->nev,nev0);

    /* első elem */
    if (*bptr==NULL)
    {
       *bptr = newptr;
       (*bptr)->ptr = NULL;
       return;
    }

    /* ha az új nev <= az első elemének, akkor beszúrjuk az
      első elem elé */
    if (strcmp(nev0, (*bptr)->nev)<=0)
    {
       newptr->ptr = *bptr;
       *bptr = newptr;
       return;
     }

    /* Megkeressük, hogy melyik két elem közé esik az nev0 */
    /* ellenőrizzük a nála nagyobbat, vagy a lista végét   */
    sptr = *bptr;            /* első    */
    tptr = (*bptr)->ptr;     /* második */
    while (tptr != NULL && strcmp(tptr->nev, nev0)<=0)
    {
      sptr = tptr;           /* továbblép */
      tptr = tptr->ptr;
    }

   /* mivel nincs az nev0-nál nagyobb elem, a lista végére tesszük
     az új elemet */
   if (tptr==NULL && strcmp(sptr->nev,nev0)<=0)
   {
       sptr->ptr   = newptr;
       newptr->ptr = NULL;
       return;
   }
```

```
    /* ha az utolsó elem nagyom az nev0-nál */
    if (tptr==NULL && strcmp(sptr->nev,nev0)>0)
    {
      tptr = sptr;
      sptr = *bptr;
      /* megkeressük az utolsó előtt álló elemet */
      while (sptr->ptr!=tptr)
         sptr = sptr->ptr;
    }

    /* az *sptr és a *tptr közé szúrjuk be az új elemet */
    newptr->ptr = tptr;
    sptr->ptr = newptr;
};

/* ---------- a lista végigjárása az elemek kiíratásával ---------- */
void ListaKiirasa(pelem bptr)
{
  pelem sptr = bptr;
  while (sptr!=NULL) {
    printf("%s\n",sptr->nev);
    sptr = sptr->ptr;
  }
}

/* ------------ a lista felszabadítása ------------ */
void ListaTorlese(pelem bptr)
{
  pelem sptr = bptr, tptr;
  while (sptr!=NULL) {
     tptr = sptr->ptr;  /* a következő elem */
     free(sptr);        /* az aktuális felszabadítása */
     sptr = tptr;       /* továbblép */
  }
}
```

A program futásának eredménye:

```
Adrienn
Anna
Julia
Lena
Natalia
```

Feladatok

1. A MINTA11_02 példát bővítse ki a négyzet és a téglalap kezelésének lehetőségével! (GYAK11_1) Használjuk fel az alábbi típusdefiníciókat!

```
enum Tipus {kor, haromszog, negyzet, teglalap};
```

```
struct Alakzat {
   enum Tipus alakzat;
   union {
     double r;
     double n;
     struct {
         double a,b,c;
     } oldalak;
   } u;
};
```

A program futásának eredménye:

```
A kor sugara: 1
A kor terulete:   3.1416

A haromszog a oldala: 3
A haromszog b oldala: 4
A haromszog c oldala: 5
A haromszog terulete:   6.0000

A negyzet oldala: 2
A negyzet terulete:   4.0000

A teglalap a oldala: 2
A teglalap b oldala: 3
A teglalap terulete:   6.0000
```

2. Készítsük el a gyorsrendezés (*quicksort*) algoritmusának rekurzív megvalósítását! A gyorsrendezés lényege, hogy áthelyezésekkel egy kiválasztott (középső) elemtől balra csak nála kisebb, míg jobbra csak nála nagyobb elemek helyezkednek el, így a kiválasztott elem a helyére kerül. Majd ugyanezt folyatjuk a két féltartományban és így tovább. (GYAK11_2)

A program futásának eredménye:

```
A rendezetlen tomb elemei :
 123   12   34   32    1    2   -3   23   79    0   -7 -19

 A rendezett tomb elemei :
 -19   -7   -3    0    1    2   12   23   32   34   79 123
```

3. Készítsünk egész számokat tároló halmazok kezelésére programot, felhasználva a karakterhalmazoknál (MINTA11_05) elmondottakat! A halmazelemek tárolására használjunk dinamikus helyfoglalású tömböt! (GYAK11_3) Vegyük figyelembe az alábbi programrészleteket!

```
typedef struct HALMAZ {
  unsigned char * p;
  int n;
} Halmaz;
```

```
/*MAIN.C*/
#include <stdio.h>
#include "halmaz.h"

int main()
{
  Halmaz a=UjHalmaz(13);  /* 0, 1, 2, ...12 */
  Halmaz b=UjHalmaz(24);  /* 0, 1, 2, ... 23 */
  Halmaz c,d,e,s;
  . . .
}
```

A program futásának eredménye:

```
A =  [    4    7   12    ]
Az A halmaz elemeinek szama: 3

B =  [    7    9   12   23    ]
A B halmaz elemeinek szama: 4

A U B =  [    4    7    9   12   23    ]

A x B =  [    7   12    ]

A \ B =  [    4    ]

12 eleme a B halmaznak

S =  [    ]
S reszhalmaza B-nek

S =  [   12   23    ]
S reszhalmaza B-nek
```

4. Alakítsuk át a neveket rendezett listában tároló példaprogram adatstruktúráját kettős listává, ahol a második lista a névhez tartozó életkor szerint láncolja az elemeket! A megoldás során használjunk több programmodult! (GYAK11_4) Alkalmazzuk az alábbi deklarációkat!

```
/* a listaelem típusa */
typedef struct ListaElem{
   int kor;                  /* életkor */
   char nev[123];            /* név     */
   struct ListaElem * korptr;
   struct ListaElem * nevptr;
} telem ;

typedef telem * pelem;

/* új elem beszúrása a listába */
void ListaUjELem(pelem * bnevptr, pelem * bkorptr,
                 const char nev0[], int kor0);

/* a lista nevsor szerinti végigjárása kiírással */
void ListaNevsorKiirasa(pelem bptr);
```

```
/* a lista kor szerinti végigjárása kiírással */
void ListaKorokKiirasa(pelem bptr);

/* a lista felszabadítása */
void ListaTorlese(pelem bptr);
```

A program futásának eredménye:

```
  Adrienn      17
     Anna      14
    Julia      29
     Lena      25
  Natalia      25

     Anna      14
  Adrienn      17
  Natalia      25
     Lena      25
    Julia      29
```

F3. A CD-melléklet tartalomjegyzéke

\Acrobat

Az ***Adobe Acrobat Reader*** telepítő-állományai:
\Windows 2000, XP*AdbeRdr602_hun_full.exe*,
\Windows 98 SE*AdbeRdr60_hun.exe*

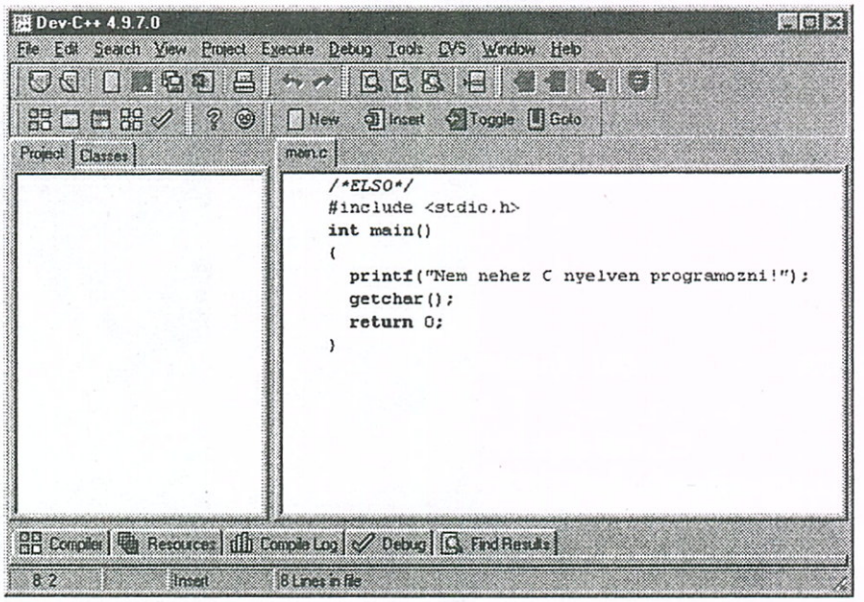
\Algoritmusok

A könyvben használt matematikai módszerek részletes leírása:

\Integral.pdf	Numerikus integrálás (numerikus kvadratúra)
\Interplacio.pdf	A függvények közelítési módszerei
\Lu.pdf	Lineáris egyenletrendszer megoldása
\Masodfoku.pdf	Másodfokú egyenlet megoldása
\Nemlin.pdf	Egyismeretlenes nemlineáris egyenlet megoldása
\Regresszio.pdf	Regresszió
\Rendezes.pdf	Rendezési módszerek

\DevCpp

A ***Dev-C++* Win98/2000/XP** alatt futtatható, szabadon terjeszthető C/C++ fordító: *devcpp4970-gcc295.exe*



\Info

A CD-melléklet tartalmával kapcsolatos információk

\Programok

A könyvhöz tartozó forrásprogramok:

\Fejezet_*XY*	Az *XY.* fejezet példaprogramjait tároló alkönyvtár
\Gyak*XY*_C	Az **F2. függelék** *XY.* gyakorlati foglalkozásának mintapéldai (*/MintaXY_...*) és feladatmegoldásai (*/GyakXY_...*).

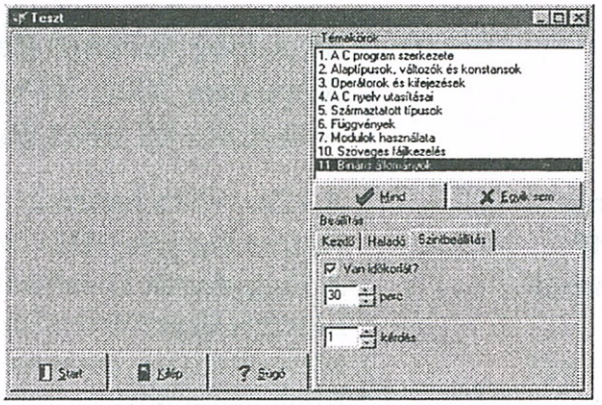
\Tanari kezikonyv

Az **F2. függelék** gyakorlati foglalkozásokhoz átdolgozott ismeretanyaga és forrásprogramjai.

\Gyakorlat_*XY*.pdf	Az *XY.* gyakorlati foglalkozás tananyaga PDF-formátumban.
\Gyak*XY*_C	Az *XY.* gyakorlati foglalkozás mintapéldai (*/MintaXY_...*) és feladatmegoldásai (*/GyakXY_...*).

\Teszt

Az önálló értékelést és tanulást segítő alkalmazás, illetve feladatlapok PDF-formátumban.

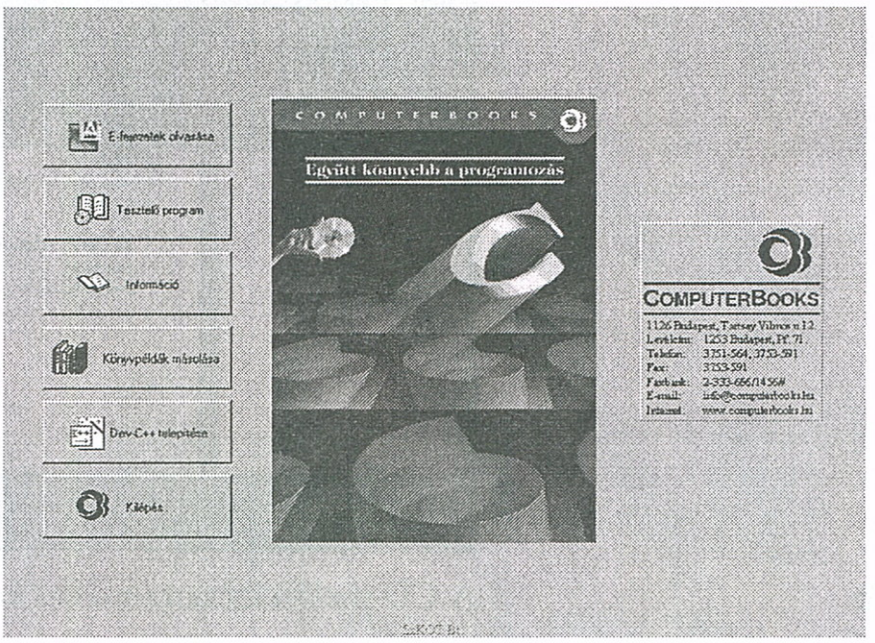


\Telepit

A könyvhöz tartozó forrásprogramok merevlemezre telepíthető formában

A CD használatát segítő segédprogramok és állományok

Fontos.exe



CDTartalom.exe

Fontos.htm

Tárgymutató

#

#define, 208
#include, 18, 250

A

abs(), 50
adatfeldolgozás, 101
adatkódolás, 29
adatrejtés elve, 5
algoritmusok, 95
aritmetikai operátorok, 166
asszociativitás, 26
átdefiniált adattípus, 60
automatikus konverzió, 186

B

balérték, 16
balra igazítás, 8
beágyazott struktúra, 65
bemenő paraméter, 198
bináris állomány, 119
binary, 23
biteltoló művelet, 30, 173
bitműveletek, 29, 173
blokk, 37
break, 42, 47, 177, 181
buborékrendezés, 104

C

calloc(), 55, 187
cast, 19, 167
char, 15
char tömb, 59
ciklusutasítás, 45, 180
const, 7, 16, 165
continue, 47, 181
CR/LF, 111
ctype.h, 176

D

decrement, 172
definíció, 7, 198
deklaráció, 7
delete operátor, 187
dinamikus helyfoglalás, 63
direkt módszer, 143
do - while, 181
double, 7, 17
double*, 17
do-while, 50

E

egész konstansok, 165
egész szám, 7
egyoperndusú operátor, 23
egyszeres pontosságú lebegőpontos szám, 7
ekvidisztáns, 135
előfordító utasítás, 3
elsőbbségi szabály, 23
elsődleges kifejezés, 23
End Of String, 58
enum, 14, 61
EOF, 12, 111, 227
EOS, 12, 58
error, 3
értékadás, 27
értékvesztés, 167
extern, 86, 166, 250

F

fájlmutató, 111
fastruktúra, 108

fclose(), 111, 227
fejállomány, 166
fejlesztői környezet, 4
felsorolt típus, 14
feltételes operátor, 32, 175
feof(), 112, 227
fflush(), 111, 227
fgetc(), 113, 228
fgets(), 229, 239
FILE*, 111, 227
fopen(), 112, 119, 228, 237
for, 48, 49, 59, 117, 180
formátumsztring, 4, 8, 10, 166
főátlóbeli elem, 57
fprintf(), 113, 231
fputc(), 113, 228
fputs(), 113, 229, 239
fread(), 120, 237
free(), 18, 34, 56, 61, 63, 187, 193
fscanf(), 231
fseek(), 120
ftell(), 120, 238
függvény, 67, 198
függvény paraméterlista, 198
függvényre mutató pointer, 209
fwrite(), 120, 237

G

getchar(), 4, 12, 161, 176, 182
gets(), 12, 59
goto, 181
gyorsrendezés, 106

H

halmaz, 259
hozzáadás a lánc elejére, 217
hozzáadás a lánc végére, 218

I

if utasítás, 37
implementációfüggő, 31
implicit konverzió, 186
increment, 172
index, 53
int, 7
int*, 20
int**, 20
interpoláció
  Lagrange, 139
isdigit(), 176
islower(), 176
isupper(), 176
iteratív módszer, 143

J

jobbérték, 16

K

karakter, 7
karakterhalmaz, 259
karakterkonstansok, 165
karaktersorozat, 58
keresési algoritmus
  bináris, 101
  szekvenciális, 101
két operandusú operátor, 23
kétdimenziós tömb, 57
kétirányú láncolt lista, 217
kétszeres pontosságú lebegőpontos szám, 7
kifejezés, 37
kifejezésutasítás, 37
kimenő paraméter, 198
konstans, 35
közvetlen beszúrással történő rendezés, 103

L

láncolt lista, 89, 215
lebegőpontos konstansok, 165
legkisebb közös többszörös, 98
legnagyobb közös osztó, 98
léptető operátorok, 27, 172
lineáris egyenletrendszer, 142
listakészítés, 263
logikai operátorok, 175
long, 7, 14
long double, 11, 31
LU-dekompozíció, 143
lvalue, 16

M

main függvény paraméterei, 82
main(), 3, 67, 155, 166
makrók, 208
malloc(), 18, 34, 55, 61, 63, 90, 186, 187, 192
matematikai függvények, 20
math.h, 20, 35, 50
mátrix, 57
mátrixinvertálás, 152
megjegyzések, 165
mellékhatás, 26
memóriafoglalás, 7, 18
mezőszélesség, 168
moduláris programozás, 5
mutató, 17, 185
mutatókonstans, 165

N

nem látható jelek, 165
Newton-Cotes féle kvadratúramódszer, 135
Newton-Raphson-módszer, 131
NULL, 12, 18, 56, 63, 90, 187, 188, 222
numerikus integrálás, 134
  kis Simpson-formula, 136
  összetett Simpson-formula, 137
numerikus módszerek, 131
nyelv alapelemei, 165
nyíl operátor, 192

O

operandus, 23
operátor, 23

P

paraméterek, 68
  felhasználói típusú, 71, 72
  sztring, 79
  tömb, 75
  változó hosszúságú, 81
paraméterlista, 67
parancsikon, 155
pointer, 17
postfix, 23, 59
pow(), 35
precedencia, 23, 25, 166
prefix, 23, 28
primary, 23
printf(), 3, 7, 166, 168
projekt, 5, 155, 250
prototípus, 3
putchar(), 12
puts(), 12

Q

quicksort, 106

R

register változó, 166
rekurzió, 256
rekurzív függvény, 83
relációjelek, 175
rendezés, 103
rendezés cserével, 103
return, 3, 42, 67, 70, 198, 218
rewind(), 120, 238
rövidzár hatás, 41
rvalue, 16

S

saját típus, 15, 36
scanf(), 7, 10, 58, 59, 113, 166, 169
Shell-rendezés, 104
short, 7, 11, 14
short int, 29
side effect, 26
sizeof(), 31, 54, 174, 194
Standard Input Output, 3
static, 250
static változó, 166
stdin, 12
stdio.h, 3, 7, 111, 161, 227
stdlib.h, 18, 55
stdout, 12
strcmp(), 59, 114
strcpy(), 59, 167, 193
string.h, 59, 114, 167
strlen(), 114

struct, 61, 192
struktúra, 7, 61, 192
struktúra átadása függvénynek, 205
struktúratípus, 61
struktúratömb, 62, 74
switch, 41, 83
switch utasítás, 177
szám összes osztói, 95
szám törzstényezői, 97
szövegfájlok, 111
sztring, 58
sztring konstans, 58
sztring literálok, 58
sztring másolás, 59
sztringkezelő függvények, 167
sztringkonstansok, 165

T

tárgymodul, 5
térbeli vektorok kezelése, 250
típus nélküli általános mutató, 186
típuskonverzió, 19, 186
típusmódosítás, 33, 167
tokenek, 165
több modul, 250
több modulból álló program, 86
többdimenziós tömb, 56
többirányú elágazatatás, 177
többszörös indirektségű mutató, 20
tömb, 53
tömbök átadása függvénynek, 201
tömbváltozó, 187
törlés a lánc közepéből, 218
törzstényezőkre bontás, 98
typedef, 15, 192

U

unary, 23
unió, 7
union, 253
unsigned, 14
utasítás, 37

V

végigpróbál módszer, 98
vessző operátor, 32
visszatérési érték, 68
void, 53, 61
void*, 18, 19, 56, 186, 187
volatile, 7, 165

W

warning, 3
while, 45, 48, 117
while utasítás, 180
whitespace, 11, 165

Z

zárójelezés, 24

Megrendelőlap

info@computerbooks.hu
Telefon. 3751-564, 3753-4591

... *Dr.Kovács T.– Dr.Kovácsné C. J.– Ozsváth M.:* **Adatkezelés az MS ACCESS 2000 alkalmazásával**	**3.497.-**
... *Kovács L.:* **Adtatbázisok tervezésének és kezelésének módszertana**	**4.200.-**
... *Mudri István:* **Digitális videózás Adobe Premiere 6.0 alkalmazásával - Win/Mac**	**3.400.-**
... *Pintér M.:* **AutoCAD 2004 felhasználói ismeretek**	**4.919.-**
... *Pintér Miklós:* **AutoCAD 2002 + LT, tankönyv és példatár 1. kötet - síkbeli rajzok**	**3.900.-**
... *Pintér Miklós:* **AutoCAD 2002 tankönyv és példatár 2. kötet - térbeli ábrázolás**	**4.500.-**
... *Móricz Attila:* **Tippek a CD íráshoz - ◎**	**2.500.-**
... *Benkő L.–Benkő T.né–Tóth B.:* **Programozzunk C nyelven – ◎**	**2.968.-**
... *Benkő T.né–Poppe A.*: **Együtt könnyebb a programozás - C – ◎**	**3.220.-**
... *Tóth B.:* **Programozzunk C++ nyelven ! ◎**	**2.994.-**
... *Benkő T.né–Poppe A.*: **Együtt könnyebb a programozás - Objektum-orientált C++ - ◎**	**3.217.-**
... *Kőhalmi Éva- Kőhalmi Mariann:* **CorelDraw 12**	**5.590.-**
... *László József:* **Dinamikus weboldalak programozása - 💻**	**4.500.-**
... *Kovalcsik Géza:* **EXCEL 2000 –◎**	**3.400.-**
... *Kovalcsikné Pintér Orsolya:* **Az EXCEL függvényei A-tól Z-ig**	**3.779.-**
... *László József:* **Hangkártya programozása** Pascal és Assembly nyelven – 🖫	**2.900.-**
... *Szirmay-Kalos.:* **Háromdimenziós grafika, animáció és játékfejlesztés -◎**	**5.496.-**
... *Ács Zsolt*: **Linux operációsrendszer-váltás-Linux 9 adminisztrátori ism. ◎**	**2.979.-**
... *Czenky M.-Tamás P.- Vágási J.:* **Tanuljuk együtt az informatikát ECDL elméleti modul - ◎**	**3.495.-**
... *Dr.Kovács T. –Dr.Kovácsné C.J. –Ozsváth M. –G.Nagy J.:* **Mit kell tudni? a PC-ről** - az OKJ és ECDL vizsgákhoz; Számítástechnikai alapismeretek; Windows 95-98; Office 97 szoftverek; adatbázis-kezelés; e. levelezés	**1.497.-**
... *Dr.Kovács T.-Dr.Kovácsné C.J.-Ozsváth M.-G.Nagy J.:* **Mit kell tudni? a PC-ről – Példatár**	**2.990.-**
... *László József:* **Mindenkinek az Internetről**	**1.999.-**
... *Móricz A.:* **Mobiltelefon a kezekben**	**2.990.-**
... *Molnár Mátyás:* **Lotus Notes 6 felhasználóknak** – 6.5 kiegészítéssel	**2.997.-**
... *Dr. Kovácsné Cohner J.–Ozsváth M.–G. Nagy J.:* **OFFICE 2000**	**2.990.-**
... *Dr. Kovácsné Cohner J.–Ozsváth M.–G. Nagy J.:* **OFFICE XP**	**3.498.-**
... *Móricz A.* **Programozási alapfeladatok WAP oldalakhoz és WEB lapokhoz**	**3.398.-**
... *Tóth B-. Benkő T.-né és* társai.: **Programozzunk Turbo Pascal nyelven– ◎**	**2.992.-**

... *Tamás P. – Tóth B. és* társai.: **Programozzunk DELPHI 7 rendszerben–** ◎ **3.860.-**
... *Benkő T.né.:* **Programozási feladatok és algoritmusok DELPHI rendszerben–** ◎ **2.999.-**
... *Juhász Tibor-Kiss Zsolt:* **Tanuljunk Programozni Visual Basic Script Objektumok - Web + interaktív ◎ melléklettel 3.925.-**
... *László József:* **PC hardver programozás valós és védett módban 4.720.-**
... *Tátrai T.:* **MS Project 2000 4.500.-**
... *Mudri István dr:* **QuarkXpress 5 5.500.-**
... *Dr. Hatvany Béla Csaba:* **Bit képek feldolgozása VISUL BASIC rendszerben - ◎ -** (feladok C++ és Delphi nyelven) **4.999.-**
... *Tamás P. – Tóth B. és* társai.: **Programozzunk Visual Basic rendszerben–◎ 3.500.-**
... *Benkő T.né.:* **Programozási feladatok és algoritmusok Visual Basic rendszerben– ◎2.999.-**
... *Stolnicki Gyula:* **SQL programozóknak ◎ 4.970.-**
... *Kovács Péter:* **Számítógép-hálózatok Praktikus útmutató hálózatépítéshez Internet beállításokhoz, Vezetékes és vezeték nélküli hálózatokhoz 2.994.-**
... *Szerk.:Hetyei J.:* **ERP rendszerek Magyarországon a 21. században 6.900.-**
... *Szerk.:Hetyei J.:* **Vezetői döntéstámogató és elektronikus kereskedelmi rendszerek Mo-n 4.900.-**
... *Szerk.:Hetyei J.:* **Pénzintézetek és állami intézmények információs rendszerei Magyarországon. 5.500.-**
... *Gottdank Tibor*: **WEBszolgáltatások - ◎ ◎ 4.995.-**
... *Móricz Attila:* **WEBDESIGN a gyakorlatban 2.399.-**
... *Móricz A.:* **WINDOWS tippek az újratelepítéshez 2.900.-**
... *Gerő J.–Reich G.:* **WORD for WINDOWS 6.0** – magyar & angol **1.499.-**
... *Gerő Judit:* **WORD 2000 I. kötet** – kezdőknek és haladóknak **2.600.-**
... *Gerő Judit:* **WORD 2000 II. kötet** – haladóknak **3.499.-**

Gazdasági informatika:
... *Lévayné Lakner Mária:* **EXCEL táblázatkezelő a gyakorlatban** – 🖫 **1.200.-**
... *Balogh Gábor:* **Visual Basic és EXCEL programozás** – 🖫 **1.900.-**
... *Csala Péter–Csetényi Athur–Tarlós Béla:* **Informatika alapjai 3.999.-**
... *Czenky Márta:* **Adatmodellezés . SQL és Access alkalmazás** – ◎ **3.900.-**

A Weboldalunkon részletes ismertetést talál könyveinkről:
www.computerbooks.hu

Megrendelő neve
Szállítási cím **Irányítószám, város**

PROGRAMOZZUNK... sorozatunk

Ára: 3220,–Ft

ISBN 963618325-2
9 789636 183257

www.computerbooks.hu